TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TONI LAMMI

FEASIBILITY OF APPLICATION CONTAINERS IN EMBEDDED REAL-TIME LINUX

Master's Thesis

Examiner: professor Timo D. Hämäläinen

The examiner and topic of the thesis were approved on 2 May 2018

ABSTRACT

TONI LAMMI: Feasibility of Application Containers in Embedded Real-Time Linux
Tampere University of Technology
Master's thesis, 55 pages, 5 Appendix pages
September 2018
Degree Programme in Electrical Engineering
Major: Embedded Systems
Examiner: professor Timo D. Hämäläinen

Keywords: IoT, real-time, Linux, embedded, container, virtualization, Docker

Virtualization offers many benefits like information security and compatibility issue fixing. This technology is, however, not traditionally used in embedded systems due to the overhead it produces. Today's embedded systems have more processing power than the systems in the past and container technology provides a low overhead solution for virtualization making virtualization possible for embedded devices.

The aim of this thesis was to study the feasibility of container-based virtualization in embedded real-time Linux. The test system was an embedded real-time Linux device used for Internet-of-Things solutions. The container technology used in the thesis was *Docker*. Docker was chosen as it has a comprehensive documentation and has been broadly adopted by the industry.

The feasibility was studied by measuring the performance and comparing these results to other studies and reading information security and software management documentation. The performance measurements considered POSIX inter-process communication latency, memory consumption and mass storage usage. The measurements were performed with both containerized and non-containerized systems to contrast the results. Information security and software management aspects were studied by inspecting documentation.

The measurements showed that the latency is not affected by the virtualization, memory consumption is slightly larger and the mass storage usage is considerably larger and should be taken into account when planning the virtualization. Containers were also perceived to help with information security and make software management easier. The information security of the containers should not, however, be carelessly trusted.

TIIVISTELMÄ

TONI LAMMI: Ohjelmistokonttien soveltuvuus sulautetuun reaaliaika-Linuxiin
Tampereen teknillinen yliopisto
Diplomityö, 55 sivua, 5 liitesivua
Syyskuu 2018
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: professori Timo D. Hämäläinen

Avainsanat: esineiden, internet, reaaliaika, Linux, sulautetut, kontti, virtualisointi, Docker

Virtualisoinnilla on monia etuja, kuten tietoturva ja yhteensopivuusongelmien korjaus. Tätä teknologiaa ei kuitenkaan ole perinteisesti käytetty sulautetuissa järjestelmissä sen tarvitsemien resurssien takia. Nykypäivän sulautetuissa järjestelmissä on enemmän laskentatehoa kuin vanhemmissa laitteissa ja ohjelmistokonttiteknologia mahdollistaa virtualisoinnin pienellä resurssien käytöllä, mikä mahdollistaa virtualisoinnin sulautetuissa järjestelmissä.

Tässä diplomityössä tutkittiin ohjelmistokonttipohjaisen virtualisoinnin soveltuvuutta sulautettuun reaaliaika-Linuxiin. Testijärjestelmänä toimi esineiden internetissä käytetty sulautettu reaaliaika-Linux-laite. Työssä käytetty ohjelmistokonttiteknologia oli *Docker*. Docker valittiin, koska sillä on kattava dokumentointi ja se on teollisuudessa laajasti omaksuttu teknologia.

Soveltuvuutta tutkittiin mittaamalla järjestelmän suoritustasoa ja vertaamalla tuloksia muihin tutkimuksiin sekä lukemalla tietoturva- ja ohjelmistonhallintadokumentaatiota. Suoritustason mittaukset toteutettiin mittaamalla POSIX:n prosessien välisen kommunikoinnin viivettä, muistin kulutusta ja tallennustilan käyttöä. Mittaukset toteutettiin sekä ohjelmistokonteilla että ilman niitä, jotta tuloksia voitaisiin verrata. Tietoturvaa ja ohjelmistonhallintaa arvioitiin tutkimalla dokumentaatiota.

Mittausten perusteella voidaan päätellä, että virtualisointi ei vaikuta prosessien välisen kommunikoinnin viiveeseen, muistin käyttö kasvaa hieman ja tallennustilan käyttö kasvaa huomattavasti, mikä tulisikin ottaa huomioon virtualisointia suunnitellessa. Ohjelmistokonttien myös todettiin auttavan ohjelmiston hallinnassa ja tietoturvassa. Konttien tietoturvaan ei kuitenkaan tulisi luottaa varauksettomasti.

PREFACE

First, a big thank you for Wapice for its supportive atmosphere for this thesis. I was given an impression that my employer really supports my thesis and wants me to graduate as quickly as possible. I was also able to use a large part of my working time with the thesis.

I would like to thank Magnus Armholt for guidance with the thesis, Lari Rasku for giving technical feedback, Sami Pietikäinen for working as an encyclopedia for anything Linux related and Iiro Vuorio for fixing the many spelling mistakes in the thesis.

Also a big thank you for TTEPO for the countless days (and evenings) not spent studying making this happen far later than expected.

Tampere, 23.7.2018

Toni Lammi

CONTENTS

LIST OF FIGURES

TERMS AND ABBREVIATIONS

Balena	Container engine project based on Docker
BusyBox	Collection of small Linux command line utilities
Container	Virtual machine executing code on native kernel
Control Group	Linux feature for hierarchical resource management
Copy-on-Write	Data is copied only after changes are made
Docker	Container engine project
GNU	Unix-like operating system
Go	Compiled programming language by Google
hardening	The act of removing extra permissions from system
image	A template for a container
init process	The first process in a namespace
Linux	Open source operating system kernel
LXC	Container Engine
mutex	Process synchronization variable
POSIX	A group of standardized operating system interfaces
Real-Time Linux	Linux with PREEMPTIVE_RT patch enabled
Real-Time System	System where computation has to be performed before a deadline
rkt	Container engine
semaphore	Inter-process communication variable
UBIFS	File system format optimized for unmanaged flash memory
Volume	Shared file system between a container and a host
Yocto	Linux building tool

cgroup	Control Group
CLI	Command Line Interface
COW	Copy on Write
CPU	Central Processing Unit
IPC	Inter Process Communication
KiB	kibibyte, 2^{10} bytes
LSM	Linux kernel safety module
MAC	Mandatory Access Control
MiB	Mebibyte, 2^{20} bytes
OCI	Open Container Initiative
OS	Operating system
OTA	Over-the-air
PID	Process Identifier Number
UID	User Identifier Number
UTS	Unix Time-sharing System
VFS	Virtual File System

1. INTRODUCTION

Containers are a light way of virtualization and are commonly used in software systems to provide easy software management. They e.g. help deploying new software versions easily, help with information security and provide an easy way to solve conflicts among applications running in the system. With containers an application and its dependencies can be packaged into, and deployed as, a single package commonly referred to as an *image*. Containers, which are then executed based on these images, use native operating system features to isolate applications running inside them from the host system providing extra security. With containers it is also possible to fix dependency conflicts, e.g. use two different versions of a library in a system: different images can have different version of the library in question.

Using containers of course introduces overhead to the system: providing two different versions of a library consumes approximately two times the space of a single library. Additionally, running software inside containers needs more operating system (OS) structures to be used to provide the virtualization for the application which consumes more memory. Moreover, initializing containers requires time and start up times might be longer than with natively run software.

With higher end computers such as servers and Personal Computers the overhead introduced by containers is quite minimal compared to the benefits. With embedded systems, where the system resources are often limited, the use of containers should be evaluated more carefully.

The aim of this thesis was to study if containers are a feasible technology for embedded and real-time systems and what are the advantages and disadvantages of using them. The properties used in evaluation were inter-process communication (IPC) latency, memory consumption, mass storage usage, information security and software management. The first three properties were mainly evaluated using measurements and the rest by studying the Docker documentation. Additionally, studies considering container technologies were inspected. IPC latency was measured using multiple processes running either natively or inside container interacting with one another via POSIX IPC. This was chosen as a method due to easy implementation and small number of variables affecting the measurements. Memory usage was measured simply by executing the above-mentioned IPC latency test and at the same time logging the output of *free* program. Mass storage usage was studied by inspecting the size of files included in the build by the programs needed by containers. The container technology chosen for the evaluation was *Docker* which was chosen due to already existing *Yocto* recipe, widespread adoption in industry and comprehensive documentation.

Chapter 2 explains the term *virtualization* and provides an overview of the virtualization technologies. Chapter 3 discusses the Linux features that consider containers and chapter 4 discusses a *container engine*, Docker, deployed into the embedded Linux system. Chapter 5 explains the structure of the tests and how they were executed on the system followed by chapter 6 discussing the results gained from those measurements. Finally, chapter 7 concludes the results of the study and discusses the possible continuations for this thesis.

2. VIRTUALIZATION

This chapter explains the term *virtualization* and discusses common virtualization technologies: *containers* and *virtual machines* (VM).

2.1 Overview

Term "virtual" can be described as an entity that does not exist yet behaves as if it did. This is very much how virtualization works in computer systems: part of the system is isolated from its surroundings and from within this part it seems to be a whole system itself. Virtualization can be implemented in multiple ways. VMs for example emulate computers being therefore virtual computers and allow executing software on a software layer. Containers use the features of the native kernel to isolate processes running in them from parts of the system while the processes themselves are executed without an extra software layer added between the processes and hardware.

The virtualization generally makes the system more secure as an attacker getting access to a virtualized environment does not have access to the whole system. It also allows fixing compatibility issues among different applications running on a computer but of course has also disadvantages. The major one is that it produces overhead to the system since some of the resources of the system need to be used for managing the virtualization features.

2.2 Containers versus Virtual Machines

Containers offer a light way of virtualization and use the native OS features to provide this. Containers are basically sets of these features which limit the permissions and resources of the processes inside them. Performing virtualization with containers allows minimal amount of resources to be used for managing the virtualization and more allocated for the actual applications.

VMs execute a whole OS with most of its services, such as scheduler. This basically means that the system has a scheduler running first on the host machine and then on each virtual machine consuming resources. Comparison of VMs and containers can be seen in figure 1. In a container system the applications, marked with App, run directly on top of a kernel while containers, marked with C, only hide parts of the operating system. In VM system *hypervisors* run on a host OS each having their own guest OS with their own services and applications. Hypervisor is the part of the system which provides the layer between the host and the guest OSs.

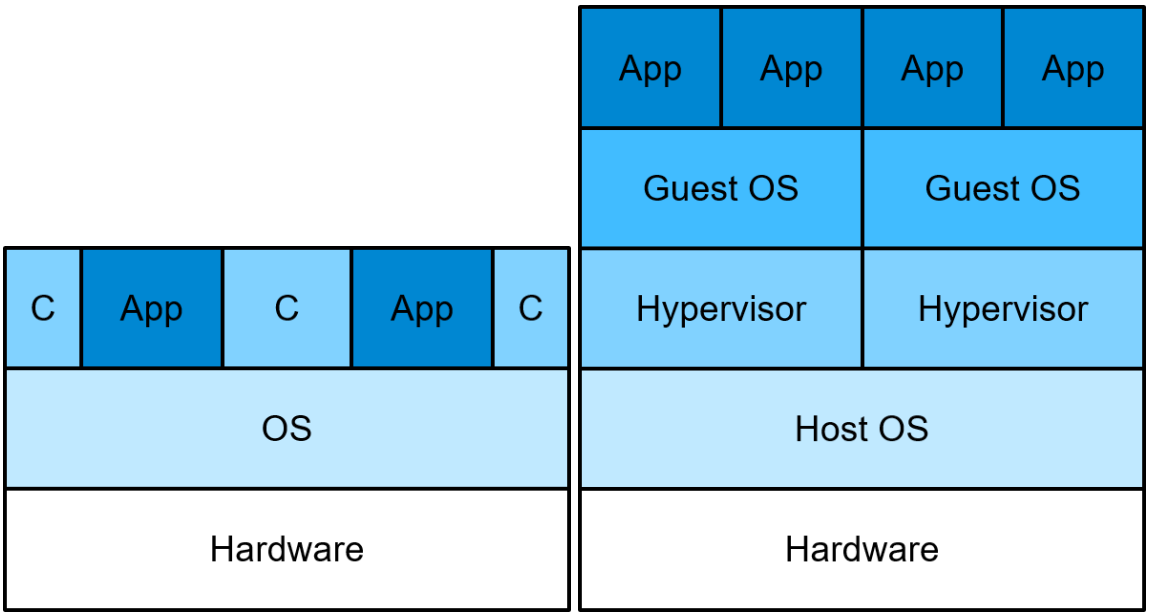

Figure 1. *Comparison between containers and virtual machines*

When comparing VMs and containers, containers excel in overhead and customization. The maximum number of containers in a system is far larger than the number of VMs and *Docker* actually proposes running one service per container [1]. Isolation between a container and a host system can also be customized: containers can be completely isolated from the surrounding system or the isolation can be nonexistent allowing programs inside the container to behave like normal processes in the host system.

VMs are better than containers in terms of information security. As the host and guest systems share next to no parts it is harder to harm the host machine from inside the guest machine. For example, kernel vulnerabilities cannot be exploited. VMs are also not limited to same kernel as the host unlike containers, e.g. VM running on a host Windows can have a guest Linux. Containers with Linux binaries inside can be executed on Windows only by adding a VM in between. Only restriction with VMs is that the binaries often have to run on the same processor architecture as the hardware provides.

2.3 Container Engines

Containers themselves are quite simple structures but managing them could prove to be hard if the number of containers in the system grows. Therefore, the containers should be managed using *container engines* i.e. applications meant for managing these software environments. Container engines are rather numerous but some examples are *Docker, LXC, Balena* and *rkt*.

Container industry is highly open sourced, and it is not uncommon to find container engines that support running Open Container Initiative (OCI) based images and containers. OCI is a project started by Docker that aims to standardize how containers should be pack-

aged and executed [2]. OCI currently consists of two parts: runtime-spec and image-spec. The former specifies how to run a file system bundle when it is unpacked on disk and the latter how container templates, images, should be constructed. If two container engines support these specifications, e.g. Docker and rkt, their images are compatible with one another. For example, images built for Docker can be pulled from registry and run using rkt.

Docker is a broadly adopted container engine that has been in development since 2013 and is discussed in more detail in chapter 4 [3]. Its broad usage in industry, comprehensive documentation and already existing Yocto recipe were some of the reasons it was chosen as the technology for this thesis. Yocto is also briefly discussed in chapter 4.

LXC, abbreviation of *Linux Containers*, was released in 2008 and was the underlying technology used by Docker versions prior to 1.0 [4] [5] [6, pp. 121]. Initially LXC used one lxc process for each started container for managing them and the user had to start each container directly from Command Line Interface (CLI) which might prove complicated if the number of containers in the system grew. Newer versions of LXC also provide a Docker-like daemon process LXD which can be used for managing the containers via one process.

Balena is an open source project forked from the open source version of the Docker and the source code still has a lot of references to Docker [7]. Balena is aimed more for embedded systems than Docker and tries to achieve this by reducing the size of the binaries, minimizing the usage of memory during image pull and using deltas[1] instead of complete images when downloading new versions of images. Balena was considered as a technology for this thesis but was not chosen as it is still quite young and does not have a pre-made Yocto recipe.

rkt is, again, an open source project developed for running on CoreOS, a Linux distribution meant for executing containers [8]. rkt was started in December 2014 and uses its own format of appc container specification but can also execute Docker images.

[1] a difference between two data sets.

3. LINUX PROCESS MANAGEMENT

This chapter discusses Linux resource management, permissions, namespaces and scheduling. Resource management is a Linux feature to limit access rights of processes to e.g. memory. Resource management method covered in this chapter is control group (cgroup). Linux permission management is by default user specific but can be changed to process specific using mandatory access control (MAC). Namespaces provide a light way of virtualizing Linux user space and scheduling provides the OS the means for dividing Central Processing Unit (CPU) time among the processes. All these features play a key part in managing process execution context and are also the methods used by containers to manage processes inside them.

Linux might well be the most common OS kernel in the world as it is used in devices spanning from higher end embedded devices such as WRM 247+, to cellphones, servers and supercomputers [9][10][11][12]. One of the reasons for the popularity of Linux is without a doubt it being an open source project and therefore relatively easy for anyone to access and adopt.

3.1 Control Group

Control group is a Linux feature for managing system resources and has two versions: v1 and v2 [13]. According to Linux documentation cgroup is used when referring to the whole feature or one control group, plural form is used when explicitly referring to multiple control groups and the abbreviation is newer capitalized [14]. The initial version of cgroup has existed since Linux 2.6.24 and the second version was first released as experimental in kernel 3.10 and made official in kernel 4.5. Control groups are managed via virtual file system (VFS) which is mounted typically to */sys/fs/cgroup/*. The structure of the VFS is different among the two versions of the cgroups but both provide a hierarchical structure and use *subsystems*, aka *resource controllers* or in short *controllers*, for resource management. In cgroup the parent cgroup also limits the resources of child cgroups. Specific behavior depends on the controller.

Version 1 is far more common than version 2 as it is more mature and supports more controllers but, in the future, the second version is probably become more common due to the simpler design and therefore simpler management. The user often does not have to manage cgroup directly as this is performed by other programs such as *Docker* discussed later. Only property of different cgroup versions seen by the user is that some programs might be incompatible with other version. It is, however, possible to mount both VFSs at the same time making it possible to use both versions as far as the VFSs do not share

any controllers (e.g. cpu controller cannot reside in both hierarchies simultaneously) [14]. The largest differences between the two versions are that v2 handles all controllers in one hierarchy where v1 uses individual hierarchies and v2 enforces *no internal process* rule where processes can be placed only in the root and leaf cgroups.

Controllers are used by cgroups to manage system resources [13]. They can be customized and are configured to the kernel at compile time. The controllers are non-hierarchically related to each other and are structured as directory hierarchies under the cgroup file system with each directory representing a cgroup. In the hierarchy child cgroups can have less or equal amount of resources as the parent, e.g. if a parent has 10 MiB of memory allocated to it, each child can have a maximum of 10 MiB of memory allocated. Some controllers additionally limit the child cgroup resources. For example, setting maximum process count in *pids* controller cgroup sets maximum limit to whole sub tree under the cgroup but limiting CPU shares using *cpu* controller sets the maximum value of cpu shares to that same value in all descendants. Some examples of the resource controllers are:

cpu Can be used e.g. to guarantee a minimum count of *CPU shares* in case the system is busy

cpuset Can be used to map processes to certain cores

memory For limiting and reporting of process memory, kernel memory and swap memory

devices Provides a way to manage the permissions for creating, reading and writing device files

freezer May be used for suspending and restoring different process bundles

pids Limits the total number of processes that can be created in a cgroup and its children

Control group versions have small differences between controller support, e.g. cpu controller can be used for managing real-time (RT) processes in cgroup v1 but not in v2. As Linux can have multiple cgroup hierarchies mounted at the same time the controllers can be distributed between the hierarchies but can only be in one hierarchy at a time. This for example makes it possible to place all possible controllers into cgroup v2 and use cgroup v1 for controllers that are not yet supported in cgroup v2.

3.2 Namespaces

Linux namespaces provide a light way to isolate processes from parts of the OS, i.e. they are a form of virtualization. Where cgroup mainly manages the hardware resources the namespaces target the OS. Linux has seven namespaces:

- cgroup namespace
- IPC namespace
- Process Identification Number (PID) namespace
- Mount namespace
- Network namespace
- UNIX Time-sharing System (UTS) namespace
- User namespace

Additionally, GNU has *chroot* core utility which is not a namespace but can be used for isolation [6, pp. 105]. Chroot changes the root directory of the calling process but it is not counted as a namespace because it is not part of kernel and does not provide a full isolation. For example, open files are kept open and root can escape the pseudo root e.g. with commands *"mkdir foo; chroot foo; cd .."*.

Namespaces are associated with user space programs and these connections are managed with *nsproxy* structures which can be seen in program 1. Each process points to zero[1] or one *nsproxy* instance. Variable *count* is used to keep track of how many processes use the corresponding nsproxy. The rest of the variables are pointers to namespace instances to which the pointing processes belong to except for *pid_ns_for_children* which points to PID namespace given to child processes of the pointing process. User namespaces behave differently compared to other namespaces and are not managed by nsproxy structures but instead use their own functions. Schematic for connections between processes and namespaces can be found from figure 2.

```
1 struct nsproxy {
2     atomic_t count;
3     struct uts_namespace    *uts_ns;
4     struct ipc_namespace    *ipc_ns;
5     struct mnt_namespace    *mnt_ns;
6     struct pid_namespace    *pid_ns_for_children;
7     struct net              *net_ns;
8     struct cgroup_namespace *cgroup_ns;
9 };
```

Program 1. *Proxy structure for managing namespaces in Linux source code[15]*

Initially all task_structs in Linux point to a same instance of nsproxy. If a namespace of a process is changed a new nsproxy instance is created, process is set to point to the new instance and *count* variable of new nsproxy is set to 1 and value in old nsproxy is decremented by one [17].

[1] Process does not point to any nsproxies if it has already terminated but its parent process has not acknowledged the child's termination, i.e. process is in *Zombie* state.

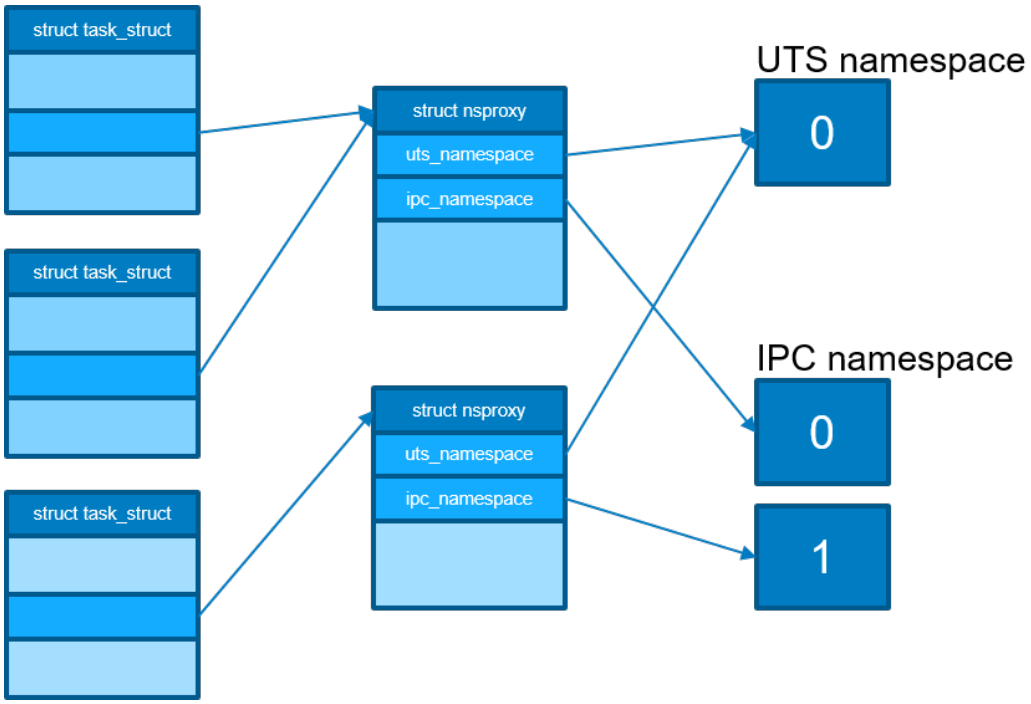

Figure 2. *Connections between processes and namespaces they are part of. Adapted from [16, pp. 50]*

Namespaces can be divided into two classes: hierarchical and non-hierarchical. Former is a parent–child hierarchy where there is one initial namespace which has zero or more children and all namespace instances can have maximum of one parent. All members of one namespace are mapped to all ancestor namespaces. In other words, the same member can have different values when viewed from different namespaces, e.g. a user can have root permissions in a child namespace but be a normal user in the parent namespace. Non-hierarchical namespace is a simple structure where all namespaces are isolated from one another and members of one namespace are not mapped to other namespaces, i.e. the members are accessible from one namespace only. Principal structures of namespace hierarchies can be found in figure 3.

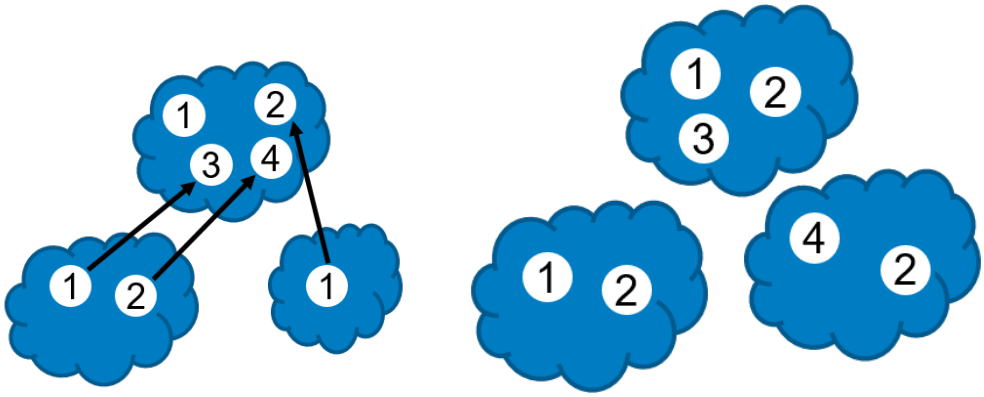

Figure 3. *Linux namespace relation types. Hierarchical on left and non-hierarchical on right*

3.2.1 Creating New Namespaces

New namespaces are typically created in Linux using functions *clone()* and *unshare()* in glibc [18] [19]. Both are wrapper functions performing the required system calls for new namespace creation. clone() creates new processes and configures them with different settings and unshare() is used by a process to modify its own execution context by creating new namespaces. Additionally glibc has *setns()* function which can be used by unthreaded processes to join already existing namespaces.

Namespaces can be created one at a time or all at once, simply by specifying more flags to the system calls. When a new process is created using clone() the possible new user namespace is guaranteed to be created first followed by other namespace instances pointing to the newly created user namespace [20].

3.2.2 UTS Namespace

UTS namespaces are non-hierarchical and simplest of the Linux namespaces. "UTS" derives from the name of the structure *struct utsname* passed to *uname()* system call which then again derives from "UNIX Time-sharing System" [21]. This term is legacy from 60's and 70's where computation was performed on large mainframes and computation time was divided into time slices among the users. Nowadays this namespace has little to nothing to do with time. UTS namespace functionality is demonstrated in program 2.

```
1 sh# domainname
2 (none)
3 sh# unshare -u
4 sh# domainname mydomain
5 sh# domainname
6 mydomain
7 sh# logout
8 sh# domainname
9 (none)
```

Program 2. *Demonstration of UTS namespace*

First command is used for checking the initial value of domain name in UTS name followed by *unshare -u* which unshares the current UTS namespace of the shell creating a new namespace. After this the command on line 4 renames the domain field of UTS namespace which is then checked on line 5. Logout is then used to exit the newly created namespace and querying domain name after this produces the initial value.

3.2.3 Inter-process Communication Namespace

IPC is a collection of OS features meant for communication between processes. These include e.g. System V shared semaphores, inter-process memory, pipes and UNIX sockets.

IPC namespace virtualizes IPC view of processes assigned to it, with exception to sockets which are managed by network namespaces [22]. Note that IPC namespaces do not affect POSIX IPC, apart from message queues. "Unsharing" rest of the POSIX IPC requires removing access to /dev/shm, e.g. by mount namespaces. IPC namespaces are quite simple structures since their relation is non-hierarchical which also implies that two processes communicating via IPC have to be in same namespace and a process whose child resides in different IPC namespace has to communicate with the child via other means, e.g. by previously mentioned UNIX sockets. IPC namespaces can be demonstrated simply with program 3.

```
1 # ipcs
2 <IPC information>
3 # netstat -a | grep "^unix"
4 <Socket information>
5 # unshare -i
6 # ipcs
7 <Empty IPC information>
8 # netstat -a | grep "^unix"
9 <Socket information>
```

Program 3. *IPC Namespace Demonstration*

The program *ipcs* shows information about IPC structures and *netstat* about sockets. The unshare call will remove all entries from ipcs listing but does not affect *netstat* as it is managed by network namespaces.

3.2.4 Process Identification Number Namespace

Linux PID namespaces are a hierarchical system where a global namespace exists on level zero containing all absolute PIDs which are also the PIDs that kernel uses for referencing processes [16, pp. 47]. When a new PID namespace is created the first process of the namespace is given the PID of 1, i.e. it is the init process of that namespace. As seen in the program 1 the value stored in nsproxy structure points to namespace given to child processes. This means that process cannot change its namespace but is able to modify into which namespace its children are placed.

The init processes have special tasks in Linux, one being that they are the parent processes of processes that are orphaned. In normal case when a process dies leaving all or some of its children alive the children are moved under the init process but this step cannot be performed when an init process of a namespace exits as there is no process under which the children could be moved. Due to this all processes in a namespace are terminated when the init process of a namespace exits. PID namespace demonstration can be seen in programs 4 and 5.

The script compares the behavior of two program calls: *unshare -fp* and *unshare -p*. In the former case unshare system call is followed by fork placing the new child in the names-

```
1  #!/bin/sh
2
3  do_child(){
4      echo "I'm child. My PID: $$"
5  }
6
7  do_parent(){
8      echo "I'm parent. My PID: $$"
9      # Unshare PID namespace and fork a new child
10     unshare -fp $0 child &
11     CHILD_PID=$!
12     echo "I'm parent. Child PID: $CHILD_PID"
13     wait $CHILD_PID
14     # Unshare PID namespace without forking
15     unshare -p $0 child &
16     CHILD_PID_2=$!
17     echo "I'm parent. Child PID: $CHILD_PID_2"
18     wait $CHILD_PID_2
19
20 }
21
22 # Execute parent if not arguments
23 # or the first argument is different than "child"
24 if [ $# -eq 0 ]; then
25     do_parent
26 elif [ $1 = "child" ]; then
27     do_child
28 else
29     do_parent
30 fi
```

Program 4. *PID Namespace Demonstration Script*

```
1  I'm parent. My PID: 6359
2  I'm parent. Child PID: 6360
3  I'm child. My PID: 1
4  I'm parent. Child PID: 6362
5  I'm child. My PID: 6362
```

Program 5. *PID Namespace Demonstration Example Output*

pace pointed by *pid_ns_for_children* in *struct nsproxy*. In the latter case PID namespace is unshared but as the unsharing only affects the child processes created after unsharing the child PID is placed in the same PID namespace as parent. This behavior can be seen in program 5 where in the first program execution the PIDs seen by parent and child are different but in the second execution the PIDs match.

3.2.5 User Namespace

User namespaces are structured hierarchically and are a parent namespace for all other namespaces as all other namespace structures have a pointer to a user namespace instance, i.e. other namespaces belong to exactly one user namespace [23].

Linux references different users using their *kuids* (kernel UID) which are not visible to user space [16, pp. 47]. Users actually have multiple different kinds of UIDs in user space but what are of interest here are *effective UIDs* of which the users have one in namespace they are part of and one for each ancestor namespace. When a new user namespace is created by clone() system call the newly created child is given all capabilities[2] in the new namespace and if the creation is performed with unshare() all capabilities are given to the current process.

By default, all namespaces apart from user namespace require root permissions for creation but since a process creating a user namespace gains all capabilities it is possible for a non-root user to create a whole set of namespaces if the user creates a new user namespace first. As the user namespace is guaranteed to be created first when specified in a system call the namespace can be created with a single system call with other namespaces. The user namespace behavior can be seen in program 6.

```
1  $id -u
2  1000
3  $unshare -i
4  unshare: unshare failed: Operation not permitted
5  $unshare -Ur
6  #id -u
7  0
8  #unshare -i
9  # echo "test" > file_owned_by_real_root.txt
10 -bash: file_owned_by_real_root.txt: Permission denied
```

Program 6. *User Namespace Demonstration*

Rows starting with $ are executed as non-root and # as root. The first unshare fails as unsharing requires root permissions except for the user namespace. After the user namespace has been unshared the unsharing of IPC namespace succeeds as the user is now a root in the new namespace. The new root is not, however, a root in a global scope which can be seen when trying to modify files belonging to the real root as in the global scope the UID is always the initial 1000.

3.2.6 Mount Namespace

Mount namespaces, also known as *virtual file system namespaces* or *file system namespaces*, control what mount points different processes see [25]. They are a useful way to control file system access for processes and work by providing different mount point lists for different processes. They were the first namespace added to Linux kernel, appearing first in version 2.4.19 in 2002. In command line, mount namespace functionality

[2] In Linux a capability is an atomic privilege to perform a task, e.g. CAP_KILL bypasses the permission check for sending signals [24]. Root processes have all capabilities.

can be demonstrated by performing commands in program 7. Mount namespace structure produced by these commands is shown in figure 4.

```
1  sh1# mount --make-private /
2  sh1# mount --make-shared /dev/sda3 /X
3  sh1# mount --make-shared /dev/sda5 /Y
4
5  sh2# unshare -m --propagation unchanged sh
6
7  sh1# mkdir /Z
8  sh1# mount --bind /X /Z
```

Program 7. *Mount namespace demonstration [25]*

In the first set of commands, run with shell number one, *root directory /* is mounted as a private mount point meaning that it and its children are not propagated into other namespaces and two other mount points */X* and */Y* are created. /X and /Y are created as shared mount points overriding the private status of the parent. The second set of commands, run on shell number 2, creates a new mount namespace (-m) copying the current namespace (--propagation unchanged) and executed sh command in the new environment. The third and final set creates a new directory and binds the sub tree found in /X to /Z. Due to private property of root mount point defined in the first set of commands, mounting of /Z is not propagated into mount namespace inhabited by shell two, but if similar bind mount would be executed in e.g. /X, the changes would propagate to 2nd namespace. It must be noted that namespaces do not affect files so modifying a file in 1st shell would be possible to see from 2nd shell, only the mount point changes are not seen, i.e. a new volume mounted in root of shell 1 would not be seen in shell 2.

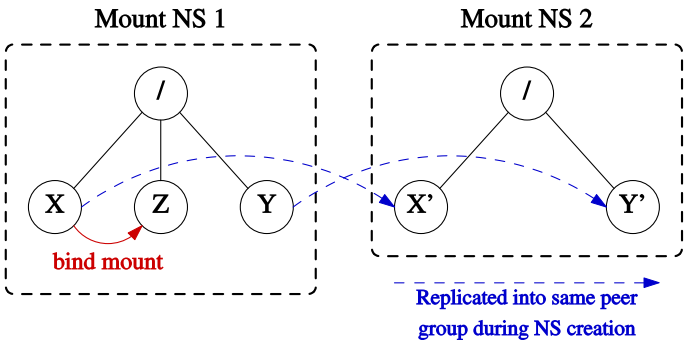

Figure 4. *Mount namespace structure produced by demonstration [25]*

3.2.7 Control Group Namespace

cgroup namespaces are used for virtualizing the view of cgroups seen by processes. In user space this means virtualizing the view of cgroups under /proc/self/cgroup. This basically works by moving virtual roots of the resource controllers down the hierarchy and works similarly with both versions of cgroup. Control group namespace itself is not

enough to virtualize cgroups. File system under the mount point of cgroup is still identical in all cgroup namespaces but instead the hierarchy is virtualized via mount namespaces [14] [26].

The main purpose for this namespace is to hide the system level view from the resource controllers and to provide better portability. Without this namespace programs running inside a virtualized execution context will still be able to gain information about the host system which is problematic for information security. Portability issues then again would rise with name conflicts, e.g. two instances of a same program running at a same time might try to manage cgroups with same names.

The namespace documentation does not seem to explicitly specify if processes in cgroup namespace can be placed into virtual root cgroups but as the virtualization works by moving the root down the hierarchy the *no internal processes rule* is probably enforced from the root cgroup namespace's point of view [14]. A demonstration of cgroup functionality can be seen in program 8.

```
1 # mkdir -p /sys/fs/cgroup/freezer/sub
2 # echo 0 > /sys/fs/cgroup/freezer/sub/cgroup.procs
3 # cat /proc/self/cgroup | grep freezer
4 3:freezer:/sub
5 # unshare -C
6 # cat /proc/self/cgroup | grep freezer
7 3:freezer:/
```

Program 8. *Simple cgroup namespace demonstration*

The first two commands create a new freezer cgroup and move the calling process to that group followed by command which checks the current freezer cgroup path. The following commands then unshare the cgroup namespace and check the freezer cgroup path in the new namespace which is now seen as a root.

3.2.8 Network Namespace

Network namespaces provide the means to create virtual networks inside Linux systems [27]. The namespaces modify how the processes see network devices, protocol stacks, IP routing tables, firewall rules, port numbers, parts of the file system considering network etc.

Physical network interfaces, such as eth0, can exist only in one namespace at a time and in case all processes in the namespace containing a physical interface are terminated, the interface is moved to parent namespace [27]. Different namespaces can be connected to each other by using virtual Internet interfaces, *veths*, which exist in pairs. One end of a veth can then be used by processes inside the namespace and other end can be connected to another interface which can be either also a virtual or a physical interface. An example network topology can be seen in figure 5.

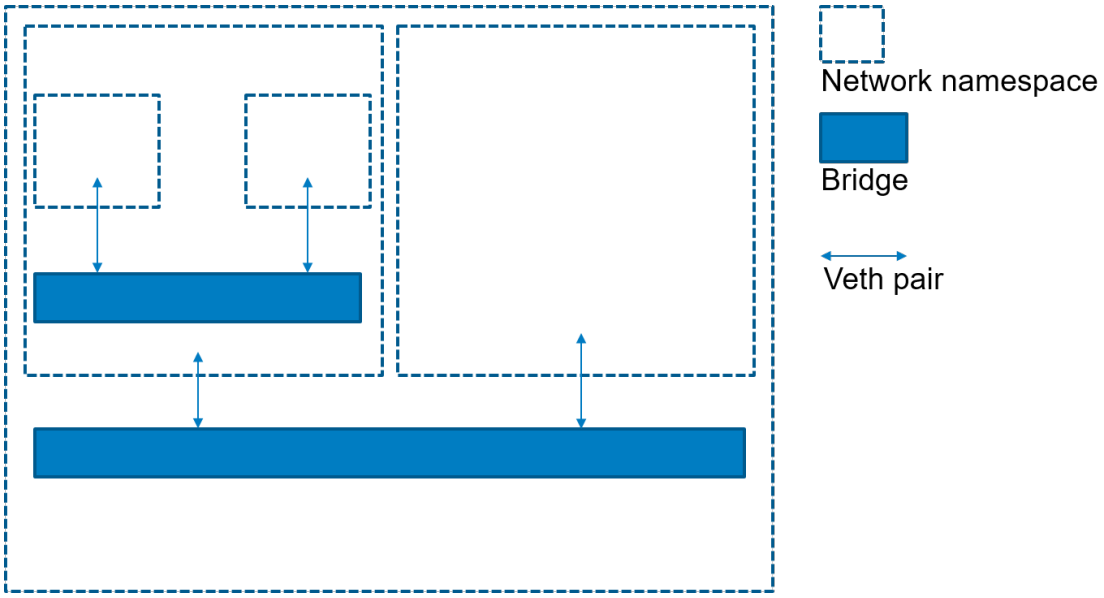

Figure 5. *Linux network namespace topology*

Network namespace functionality can be demonstrated using commands *ping 8.8.8.8; unshare -n; ping 8.8.8.8* as root which would first ping the given IP followed by changing the network namespace of the shell removing all network interfaces causing the second ping to fail. More functional example, like connecting the new network namespace to the Internet would need a lot more work like setting up routing tables, IP forwarding etc.

3.3 Mandatory Access Control

Mandatory Access Control (MAC) is a method of *hardening* the system, i.e. enhancing information security by disabling all extra process permissions. Default behavior in Linux is to manage permissions by users and groups. With this approach all the processes of the same user have same permissions and can modify all files belonging to that user. MAC mechanisms change this granularity from user to individual processes so processes of same user can have different permissions e.g. to file system. In Linux MAC is generally performed using *Linux security module* (LSM) kernel framework designed for hardening Linux systems [28]. Using LSM with addition to basic user-based access control increases system security as attacks that use e.g. code injections and buffer overflows can be more easily prevented. Some examples of kernel extensions that use LSM are Application Armor (AppArmor) and Safety Enhanced Linux (SELinux), both consisting of a kernel module and user space programs designed for enforcing MAC rules [29].

LSMs can be used for controlling process permissions to files, capabilities, network access and resource limits. This set of permissions is generally called a *profile*, which is a human-readable file found e.g. under /etc/apparmor.d if AppArmor is used. Profile files are typically quite lengthy but e.g. adding line *deny /bin/** w* would remove write permissions to /bin/ and all its descendant directories from processes with the given profile. This can also include processes belonging to root.

3.4 Scheduling

Scheduling in Linux is performed using *scheduling classes* and *schedule()* system call. In addition to these e.g. Mauerer explains scheduling with two extra entities: *main scheduler* and *periodic scheduler* [16, pp. 86,87]. These are not actually features in the kernel but ways how schedule() is called: main scheduler is called when a change happens in system processes, e.g. when a process starts to wait for semaphore or exits, and periodic scheduler is called periodically, as the name states. When the scheduler is initiated it goes through scheduling classes starting from the highest priority (deadline scheduling) to the lowest (idle scheduling) and picks the first available process [30][31].

Linux has six *scheduling policies*, from highest priority to lowest, *SCHED_DEADLINE, SCHED_FIFO, SCHED_RR, SCHED_OTHER, SCHED_BATCH* and *SCHED_IDLE*. The first three are different kinds of RT scheduling policies: processes with SCHED_DEADLINE have deadlines, processes with SCHED_FIFO are RT processes which are executed until they voluntarily give up the CPU and processes with SCHED_RR are executed using Round-Robin scheduling algorithm. Following the RT policies are the default scheduling policy used by Linux, SCHED_OTHER, which means that a process using it is scheduled using Completely Fair Scheduler (CFS) and SCHED_BATCH which is meant for non-interactive tasks which are slightly disfavored in scheduling decisions and are preempted less frequently than SCHED_OTHER processes. Finally, SCHED_IDLE is for tasks with really low priorities which are run only if no other tasks need the CPU. Linux scheduling classes are, from the highest to the lowest priority, *stop_sched_class, dl_sched_class, rt_sched_class, fair_sched_class and idle_sched_class* where stop_sched_class is used in symmetric multiprocessing systems to disable cores. The relation between scheduling policies and scheduling classes can be seen in figure 6.

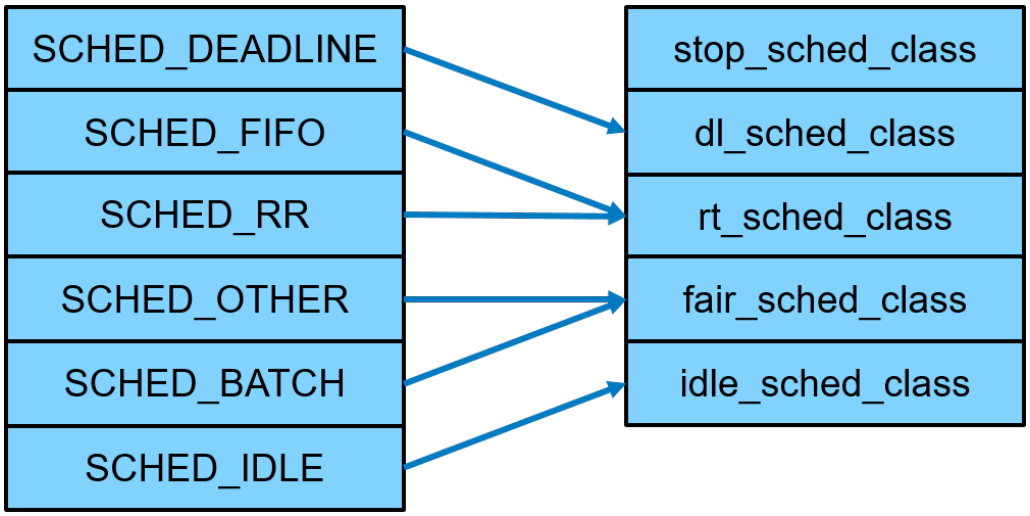

Figure 6. *Mapping between scheduling policies and scheduling classes*

Linux uses numerical process priorities ranging from -1 to 140 in parallel with scheduling classes to find out if new available tasks have permissions to interrupt currently running

tasks in kernel mode, i.e. when a task is performing system calls [32]. The mapping of priorities between kernel and user space can be seen in figure 7.

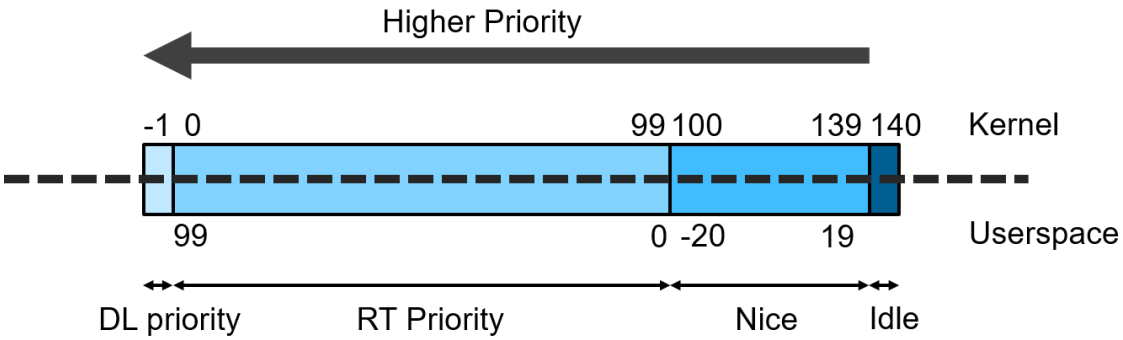

Figure 7. *Priorities of Linux processes*

In user space, deadline scheduling processes do not have priorities and according to kernel they always have static priority of -1. The actual priority itself is dynamic and calculated based on the time left before deadline. RT processes have priorities running from 0 to 99, 99 being the highest priority in user space, and in kernel these are inverted 0 becoming 99 and 99 becoming 0. Normal processes have *nice* values from -20 to 19 in user space mapping to values from 100 to 139 in kernel. *Batch* processes have the same priority range as normal processes but have longer execution time slices and are presumed as CPU-intensive. In the user space the nice values function as weights for the processes, i.e. a process with a smaller nice value will have more execution time than a process with larger nice value. Idle processes are the processes with the lowest priority, even lower than nice 19 and are executed when no other processes need execution.

3.5 Real-Time Linux

Real-Time (RT) Linux (not to be mixed with RTLinux) is a child project for Linux and not included in the Linux Git repository since it would only benefit few of the Linux users and produce a lot of extra work [33]. Instead the project is hosted on Git repository on kernel.org [34]. RT Linux is typically built by first pulling Linux project from Git followed by downloading a .patch file from kernel.org, applying this .patch file to the project and finally configuring the Linux build before building the kernel.

The aim of RT Linux is to minimize the latency of Linux kernel. This is done by making the kernel as preemptible as possible e.g. by replacing *spinlocks* with *mutexes*. This causes RT Linux not to be an ideal RTOS since hard deadlines cannot be fully guaranteed but works well for soft RT tasks. The advantage of RT Linux is the support of same system calls as normal Linux. This makes reusing code easier since non-RT tasks can often be implemented with binaries available for normal Linux.

4. DOCKER

This chapter discusses container engine Docker briefly mentioned in the end of chapter 2. The focus of this chapter is Docker version 1.12.5 as it was the version deployed to embedded Linux in this thesis and the Docker versions have major differences as the project is still quite young.

4.1 Overview

Docker is a tool implemented in *Go*, used for configuring, building, running and distributing application containers [35] [3]. A high level architecture of Docker can be found in figure 8. Different arrow types in the figure represent different operations. In other words *docker build* creates a new image, *docker pull* downloads an image into the local file system and *docker run* executes a container based on the given image. Docker has five major parts: *docker, dockerd, images, containers* and *registries*. "docker" is a binary providing a CLI for accessing dockerd, the daemon process of Docker which is the entity that implements majority of the functionality of a Docker system. Images and containers are analogous to classes and objects in programming: a class is basically a template used for building objects and similarly Docker uses images for building containers. Finally, registries behave much like version control repositories and provide databases of images that can then be pulled by Docker daemons and support tagging different versions of the images.

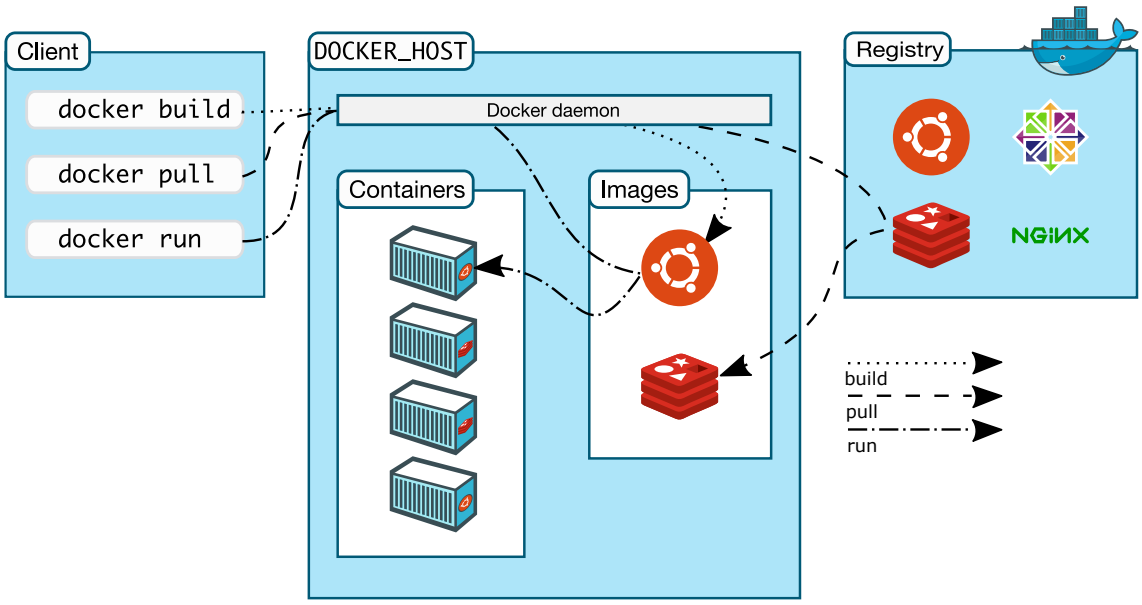

Figure 8. *Docker high level architecture [36]*

Docker has two editions: commercial Docker-EE (Enterprise Edition) and open source Docker-CE (Community Edition) [37]. Both have same core parts, but the former offers additional features such as image security scanning. Nowadays Docker supports native Windows and Mac execution but initially it was developed solely for Linux.

4.2 Architecture

A Docker system consists of multiple binaries and an example system can be seen in figure 9. docker and containerd-ctr are both executed from command line and provide a CLI to interact with daemon processes dockerd and containerd via UNIX sockets. The functionality of containerd-ctr is also implemented as part of dockerd in libcontainerd and therefore containerd-ctr is redundant in typical systems. containerd-shim is the part of the system that is given the task of supervising a single container as well as starting and stopping it via runc program. It needs to be noted that all of the programs in figure 9 can be replaced with other programs if they support OCI specification.

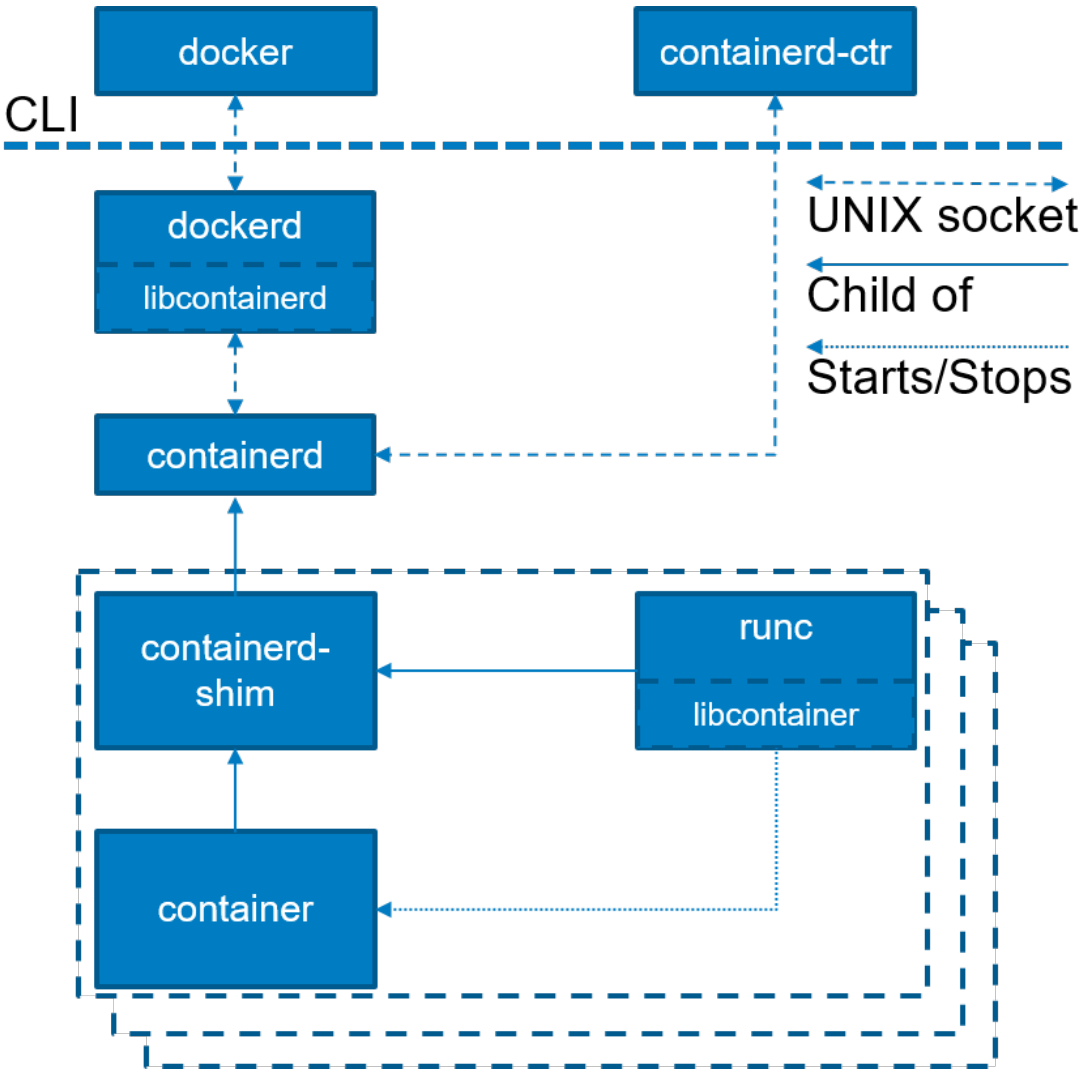

Figure 9. *Example Docker system on one computer*

Docker daemon is typically started in Linux initialization and automatically starts containerd. In Docker 1.12.5, which uses containerd version 0.2.0, containerd is only responsible of executing containers and managing running processes [38]. For example, Docker performs the mounting of container file systems. Newer versions such as containerd 1.1.0 released on 1.5.2018 perform most of the per-machine functionality [39] [40]. When a container is started, an instance of containerd-shim is created for starting and monitoring the container. containerd-shim then uses runc for starting the container. runc exits after the container has been created leaving containerd-shim as the parent of the newly created container.

4.3 Functionality

Starting a container begins with messaging dockerd to start a container, e.g. by CLI or using dockerd API directly. For example command *docker run --rm -it ubuntu:16.04 /bin/bash* uses Docker CLI to send dockerd command to execute a container based on "ubuntu" image with tag "16.04". Additional flags rm, i, and t cause container to be deleted once it exits, STDIN to be connected and pseudo-tty to be allocated for the container respectively. Finally, /bin/bash is the entry point in the container like the init program in Linux.

Once dockerd receives the command it first checks the local file system for image–tag pair downloading required layers if they are not present. After this the *union mount file system* of the container is mounted and forwarded to containerd for running. Union mount file system is discussed later in this chapter. Containerd then starts containerd-shim process for the new container, which handles the management of the container and calls runc program for interaction with containers.

Runc is the piece of the system which performs all the interactions with Linux to manage containers [41]. After the task has been executed runc exits and leaves the process calling runc in command for the newly created container to minimize the used resources. Runc process is also initiated always when changes must be made into containers, e.g. when new processes are manually started in containers.

By default, the Docker containers use UTS, IPC, PID, mount and network namespaces. User namespace can be taken into use by modifying configuration. Containers do not use cgroup namespace. This means that */proc/self/cgroup* contains information from outside the virtualized environment. Mount namespace, however, virtualizes the */sys/fs/cgroup* mount meaning that processes in container do not have access to cgroup configuration outside of container.

4.4 Features

This section discusses Docker features that should be understood when deploying Docker systems.

4.4.1 Union Mount File System

Union mount file system or *unioning file system* is a term describing series of file systems used on Linux that are based on *layers* [42]. Some of these file systems are *aufs*, *overlayfs*, *xfs* and *vfs*. Docker uses union mount file system for handling multiple images, so their total size can be kept minimal.

Union mount file system layers are simply directories under Linux file system which can then be combined into a single view. Each file system version handles the layers a bit differently, e.g. overlayfs can only handle two layers but more layers are added using hard links[1] while aufs can natively handle more than two layers. Figure 10 shows an example of overlayfs layers. Lowerdir contains all layers of the image and upperdir is the read-write layer for the container.

When a Docker container is started it is isolated from the host file system with mount namespaces. The namespace propagation set for Docker can be either *slave* or *private*. This propagation setting makes it possible to unmount the host root file system from the container root and remount the unioning mount file system. Volumes and bind mounts also work in a similar manner and are mounted after the root file system.

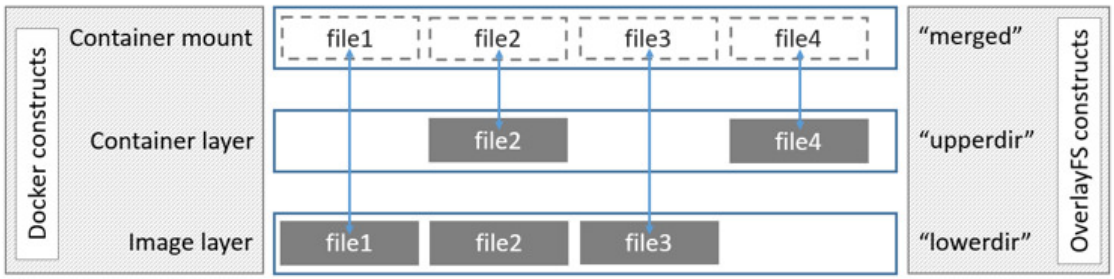

Figure 10. *OverlayFS layer schematic [43]*

4.4.2 Storage

Preferred methods for storing data with Docker containers are *volumes*, *bind mounts* and *tmpfs mounts*. The other way to store data is to store it directly into the file system of the container but as the containers use unioning file system this causes slower write speeds and data does not persist after container is deleted [44].

Volume is a part of the host file system that is shared between containers and the host and is meant to be used as an inter-container file system [45]. This part is managed by dockerd and is either removed once the container exits if *--rm* flag is specified or left dangling. Volumes can be initialized with read-only or with read-write permissions and are initialized by specifying the path inside a container that is to be mounted as a volume.

[1] Hard link is a link to an existing *inode* which is a Linux structure for managing files and directories.

If the mount path inside the container contains data preceding the mount those files are copied into the volume and are available to other containers.

Bind mounts are like volumes but instead of being managed by dockerd their mount points in the host file system are explicitly specified [46]. If the mount path of the container already contains files, those files are obscured by the files residing in the host mount point.

Tmpfs (Temporary File System) mounts work the same way as *mount -t tmpfs -o size=<size> tmpfs /path/to/dir* on normal Linux, that is, they mount the specified point directly into the memory of the machine instead of non-volatile memory [47]. This is a good way to store critical data such as passwords inside containers as the data is destroyed on power down and also provides a faster way of storage as accessing memory is many times faster than writing into or reading from a file.

4.4.3 Resource Management

Docker provides ways to limit resources such as CPU time, memory and block IO of individual containers [48]. Limits are enforced using cgroup. By default, there are no limitations except that changing scheduling policy and priorities are not enabled. Default permissions can be changed when starting containers by specifying command line arguments.

By default, containers can use CFS and all of the CPU cores but with command line arguments they can be designated to specific cores and CPU resources can be limited [48]. For example *docker run --cpus="1.5" --cpuset-cpus="0,1" --cpu-shares="512" hello-world* will run container based on image *hello-world* on 1.5 CPUs on cores 0 and 1 with weight of 512. What this means is that one of the cores behaves normally with the container and other can be allocated for container half of the time. If the CPU is overloaded --cpu-shares specifies the relative weight of the process. Otherwise it has no effect. Containers can be given permissions for executing RT processes and modifying process priorities by specifying arguments *--cap-add=sys_nice* and *--ulimit rtprio=<value>*. Where *<value>* is between 0 and 99. On the low level the flags specified in the example cause new cgroup to be created which perform the actual resource management. For example *--cpuset-cpus="0,1"* creates a new cgroup under */sys/fs/cgroup/cpuset/docker/<container hash>*. This directory then contains a file *cpuset.cpus* which has string *0-1*, i.e. the container uses cores 0 and 1.

Docker can limit container's user space memory, kernel memory, memory swap max sizes and memory swappiness[2] [48]. Additionally, it is possible to set soft limits for memory usage when Docker detects contention or low memory on host machine. For example com-

[2] Relative weight of swapping process's memory

mand *docker run --memory="4m" --memory-swap="4m" --memory-swappiness="50" --kernel-memory="20m" hello-world* will run *hello world* image based container with 4 MiB of available memory, setting max swap size to 0 bytes, setting swappiness to 50 and limiting available kernel memory to 20 MiB. Docker documentation falsely talks about Megabytes in this context when configuration JSON under container's directory functions with powers of two.

Docker currently only supports cgroup v1 as v2 does not support all required controllers such as *freezer*. The discussion for using the v2 in Docker is open as of June 2018 [49].

4.4.4 Networking

Docker offers five options for networking: *none*, *bridge*, *macvlan*, *host* and *overlay* [50]. Network interfaces of multiple containers can additionally be shared [6, pp. 94–96]. Containers with different interfaces and inter-container networks can be found in figure 11. In the OS level the networking is performed using Linux network namespaces.

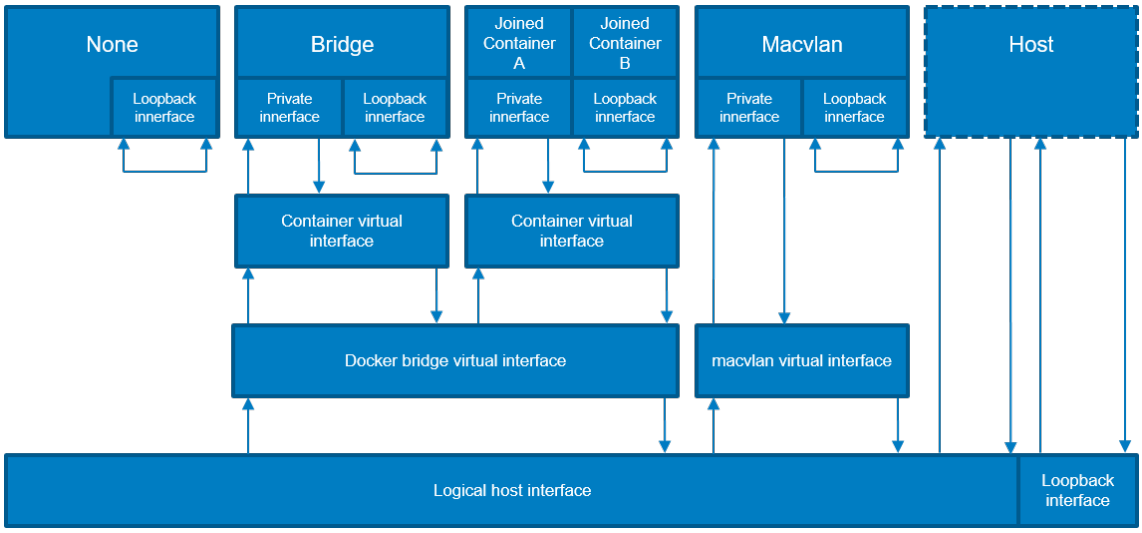

Figure 11. *Different container types. Adapted from [6, pp. 82]*

None, bridge, macvlan and host are different ways of connecting containers to host network. None is the simplest of the networking options and provides only a loopback interface without any external connections [50]. Bridge driver is the default network driver and provides a virtual bridged network inside host machine [51]. Macvlan mimics a physical network by assigning MAC address for each container in its network and can therefore be used for applications that expect to be connected directly into physical network interface [52]. Host network driver does not isolate the network stack of the container at all providing no isolation [53]. Network interface of two containers is shared in program 9. This creates a container pair where the programs of different containers are isolated from each other in all other ways except for the network interface.

Overlay network (not to be mixed with overlayfs) is a network driver operating on inter-dockerd level [54]. It provides a network connecting multiple daemons together. Docker

```
1 # docker run -d --name brady \
2     --net none alpine:latest \
3     nc -l 127.0.0.1:3333
4
5 # docker run -it --net container:brady \
6     alpine:latest netstat -al
```

Program 9. *Creating an inter-container network interface [6, pp. 95]*

versions older than 17.06 only use overlay networks for swarms (group of Docker systems providing same service) but since version 17.06 it has also been possible to connect standalone containers to overlay network [55].

4.4.5 Images and Containers

Images are file system templates stored in Docker registries bundled with metadata, e.g. default entry point for the image. Images consist of layers and each layer is named by SHA256 hash of the tarred data of that layer [56]. Docker also supports building new images based on base images. They work so that first all layers used by the base image are built and new layers are then built on top of these layers to produce a new image. This means that child images have no notion of their base images, but only of their layers. Taken these facts updating a base image does not propagate into child images without rebuilding the child images, i.e. if image B depends on image A and image A is rebuilt with new layer, image B has to be rebuilt in order for this change to propagate into image B.

Docker images are managed using URLs like *my.registry.com:5000/path/to/repository:tag* where my.registry.com is the domain name of registry, 5000 is the port and rest is used for referencing the image. In case the image is hosted in hub.docker.com the images are referenced with format *username/registry:tag*. For example, URL *arm32v7/python:2.7* references an image in Docker Hub owned by username (or in this case, organization) arm32v7, in repository python with tag 2.7 which in this case also refers to the Python interpreter version. If the tag field is omitted when dereferencing an image, the default tag of *latest* is used which in the case of this image points to same image as tags *3, 3.6* and *3.6.5*. The tags and image contents do not have to match in any way and e.g. *user/repo:tag1* and *user/repo:tag2* could have no common files at all. This is of course not recommended as the repository structure might prove complicated with larger repositories and therefore naming should be systematic e.g. one functionality per repository and tags pointing to different versions of the image. An example structure of a repository can be seen in figure 12. The columns in the figure represent the images and their layers and tags point to different instances of the same repository. As seen in the figure, tags do not have to have any common layers.

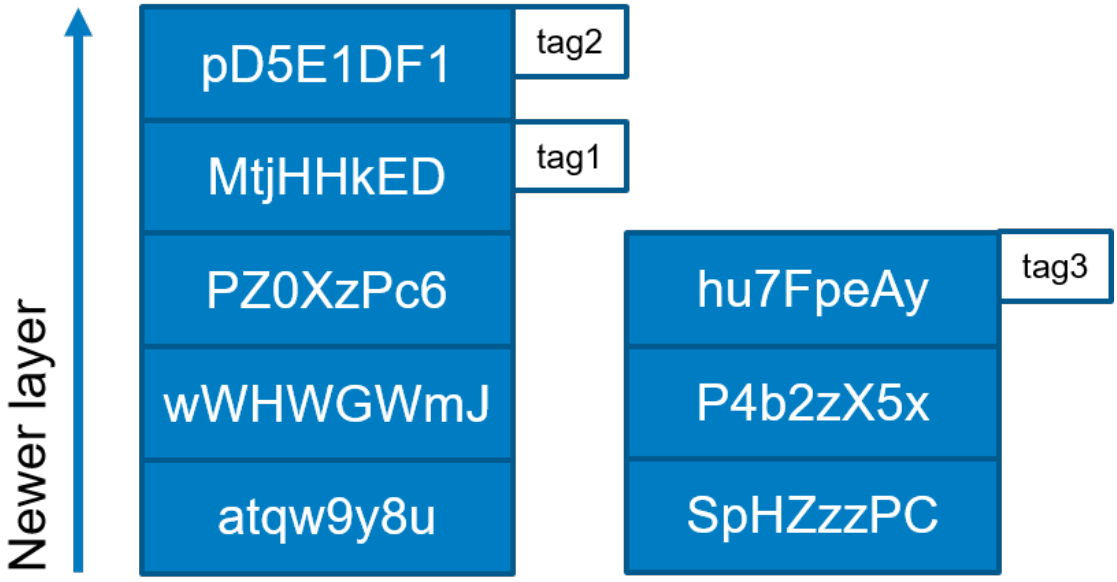

Figure 12. *The relation of tags and layers*

Containers are deployed instances of images and have multiple read-only layers on bottom, which are the same layers as in images they are based on, and one read-write layer on top of these where all the changes are made. As the structure is similar it is also possible to produce new images by storing the state of containers but overall this is not recommended as it will increase the number of layers and produce extra overhead.

4.4.6 Swarm Mode Overview

Swarm is a Docker mode used for grouping machines together where one group is called a *swarm* [57]. Swarm mode is designed to be used for servers where scalability is an important trait. Embedded systems however rarely need to offer services scalable this way and therefore swarm mode is only briefly discussed.

Swarm mode revolves around services which are defined and assigned for a swarm and can be marked as replicated or global. The former kind are deployed depending on the need and the latter kind once for each available machine which meets the placement requirements and resource constraints. A swarm works by first creating a virtual *overlay network* which puts all swarm devices under a single network, which also implies that devices have to be visible to each other. When swarm services are then requested from a certain port mapped for swarm, the traffic is rerouted to devices performing the requested service. If multiple machines with a same service exist, the requests are automatically balanced among the devices.

4.5 Information Security

One of the key benefits of virtualizing the execution context of a process is security. Containers are of course a better solution than native execution of software from the security point of view since an attacker has to cross one more barrier before accessing the whole system. This section discusses few security problems of Docker and how they can be addressed.

Older Docker versions are not able to protect the system from fork bomb attacks[3]. This is due to Linux versions prior to 4.4 not having pids cgroup. A workaround for this is to set PID limits on users but in this case if a user starts multiple containers, all processes inside these containers count towards the maximum number of PIDs. One other way is to manually put dockerd to pids cgroup for limiting the number of PIDs if the kernel version is new enough, but Docker still lacks the support for this functionality. This of course limits only the total count of processes running in containers rather than managing individual containers. Docker versions since 1.11 have *--pids-limit* flag for limiting the number of processes inside a container.

Privilege escalation is a real threat in all Docker versions since user namespace is not enabled by default and the settings can be overridden using command line arguments [58]. Any user having access to socket used by dockerd, e.g. root users or users belonging to *docker* user group, can execute a container and the same user will gain permissions of root inside that container. This alone is not dangerous but by running command *docker run –rm -it -v /:/mnt/ ubuntu /bin/bash* as user with UID 1000 I was able to modify files belonging to the root user in the host system without password checks. This command starts a Docker container and mounts the root of the host file system into /mnt/ path in the container file system. The user namespace can be enabled by modifying dockerd configuration but as the user can override default configuration by specifying command line arguments the problem cannot be removed. The only way to prevent the privilege escalation is to limit the permissions to dockerd socket.

Docker containers can be easily hardened using AppArmor profiles [59]. This profile is assigned to container at start time with options *--security-opts apparmor=profilename*. Container which do not need any access to host file system should for example be completely revoked of permissions there. Additional security can be achieved by applying a MAC profile also for dockerd. Network interfaces for the containers should also be hardened. This can be achieved with using a fitting network setting. For example, containers which do not need a network connection should use Docker network *none* i.e. isolating the container in its own network namespace without any virtual Ethernets connected.

Docker registries support TLS encryption and storing only hashes of the passwords so the information security on the server side is good but dockerds have to supply the plain

[3] Fork bomb attacks are performed using a process which forks itself over and over consuming system resources.

text credentials to the servers [60] [61]. This is problematic on embedded systems as the devices have to be autonomous, so the credentials need to be stored on the device but at the same time the devices are often physically accessible. This means that if an attacker gains access to the device the credentials are completely visible. To make this harder the system can be configured to give dockerd the credentials using STDIN instead of command line arguments, so the credentials are not visible in system log. The credentials can also be hard coded into the program to make accessing them harder. By default, the credentials are stored unencrypted in /.docker/config.json. This can be fixed by configuring Docker with external program for storing credentials. Credentials should also be device specific to make accessing other devices hard and make it possible to revoke permissions from one device if compromised. This way it is also possible to change credentials in one device without needing to update all machines.

4.6 Cross Building Docker

This section discusses cross building Docker using *Yocto*. Normal Docker build system is bootstrapped by default meaning that Docker binaries are typically built using older version of Docker. This works fine for traditional use cases such as servers but introduces problems when the aim is to cross build Docker for another architecture.

4.6.1 Yocto Overview

Yocto is a powerful tool used for configuring and building Linux systems. Yocto uses OpenEmbedded project as its core and BitBake as its build tool for building custom Linux distributions [62]. Simply put, Yocto uses series of shell and python scripts to build everything regarding the new system, including toolchain, kernel, root file system etc.

Yocto build system is rather complicated on first look as it differs quite a lot from typical Makefiles commonly used for building Linux systems. The hierarchy consists of configuration files, such as *local.conf* and *bblayers.conf* which configure e.g. the architecture used and what *recipes* are included into the build. Recipes themselves are written into files ending with .bb, .bbappend and .bbclass and tell how different components are built into the system.

Yocto works in principle by building everything needed for the target system. This includes e.g. compiler and libraries. Yocto build process consists of downloading required files from their sources, e.g. from Git repositories, configuring the files, e.g. by applying .patch files to them and building parts of the file system. Each recipe places its build outputs under its own build directory in paths equal to ones in the root file system. After building phase these directory structures are combined and root file system is produced if no conflicts are detected.

4.6.2 Docker Yocto Recipe

The Yocto recipe of Docker can be found in [63]. The recipe in tag Morty has expected runtime dependencies to containerd and runc but the recipe also lists runtime dependencies to curl, aufs-util, git, util-linux and iptables. Of these dependencies the aufs-util and git can be removed if the system uses OverlayFS instead of aufs and if pulling images from git using git protocol is not needed. Git is a quite large binary so considerable amount of space could be saved by removing it. Git and curl also seem to have been completely removed from the dependencies of the recipe in newer sumo tag of Yocto which might be due to functionality being added directly to Docker.

As stated before the typical Docker build process is executed using Docker images designed for building. For Yocto, where even the Software Development Kit is built from source, this is not an option. Therefore, the build is performed natively and all dependencies are included in the Go environment used by Yocto.

4.6.3 Linux Configurations

As Docker uses Linux kernel features to virtualize the process environments Linux also requires certain configurations to be enabled for Docker to work properly. The list of required and optional configurations is easiest to acquire by running *check-config.sh* from the Git repository of Docker-CE on the development machine if it runs a Linux based system. The list gained by executing the script with the configuration of the kernel used in measurements can be found in appendix B. As seen in the configuration Docker has multiple mandatory configurations and even more secondary dependencies. The former is what are suggested to be enabled for Docker to function normally and the latter are used for enabling different features such as parts of resource management and information security. The configuration for user namespace is also an optional configuration which is due to possible single user configuration of Linux.

4.7 Setting Up a Docker System

Docker systems can be broadly customized and especially with larger image count the systems may become quite complicated. This section discusses how a Docker system can and should be configured to handle this complexity.

4.7.1 Configuring Machines

Docker daemons can be configured by either specifying command line arguments when starting the daemon or by writing a JSON configuration file which is read during dockerd start. Of these the latter is the suggested way as it is more verbose. Linux configurations can also be tweaked to enable or disable different Docker features.

Docker daemon can be configured to use TCP socket instead of Unix socket which would enable managing the system from remote location, but this of course introduces new risks and the connection should be at least password protected [64]. This approach is however not recommended and instead an SSH connection should be used for accessing the docker CLI on the remote machine. This enables the use of authentication methods other than username–password pairs and solves an issue where remote docker CLI requires root permissions on the machine it is running in. More security can be added by enabling firewall rules e.g. via iptables to enable connections only from certain addresses.

As the embedded systems deployed to field often have to be autonomous, most of the functionality in the system should be automatic. For example, updated Docker images should be pulled from registries if new versions are available. This can be accomplished by e.g. writing a custom daemon program which gives orders to Docker daemon or by writing shell scripts which use docker for interfacing the dockerd. One example of this kind of program is *open-horizon* [65]. Open-horizon works by transmitting JSON data across network and makes it possible to register *workloads* and *microservices* into a cloud. These are then automatically deployed to devices using Docker containers after a while.

One easy way to group Docker machines together is to deploy an overlay network connecting the machines. Using overlay networks, the external user can address the whole system via a single address which is a good way to abstract layout of the actual network. Swarm mode can be useful with embedded systems when the workload is not device specific. One example of this could be a distributed database which does not have to be running on every machine.

4.7.2 Configuring Docker Registry and Managing Images

Docker uses Docker Hub as its default Docker registry, but it is also possible to deploy custom registries. This can basically be implemented in any way that supports the registry API of Docker, but the easiest and preferred way is to use prebuilt Docker image meant for this purpose: "registry". The same registry image can be used for generating encrypted credentials into host system and to run the registry itself. For test purposes the registry can be executed using unencrypted HTTP connection. TLS encryption should always be used for registries that are visible to external network. An alternative authentication to standard username-based authentication is token-based authentication. Token based authentication uses external service to provide authentication which makes it possible to manage credentials without accessing registry at all. Some other things that should be considered are creating a bind mount for storing registry data even after the container is destroyed and to deploy the registry as a swarm instead of a container which enables load sharing among multiple servers if the usage is high.

As stated before, Docker images are based on layers. Modification of a base image does not propagate into child images. Therefore, modifying base image should result in recompilation of all descendant images. This recompilation step can be evaded if the base

image is changed to configuration image, which deploys configuration files to a volume which is then shared with containers which need these configurations. This multi-image solution is good for lowering the server load and bandwidth usage but might introduce a small startup latency in the target system. Figure 13 shows different structure of base images and configuration volume structure.

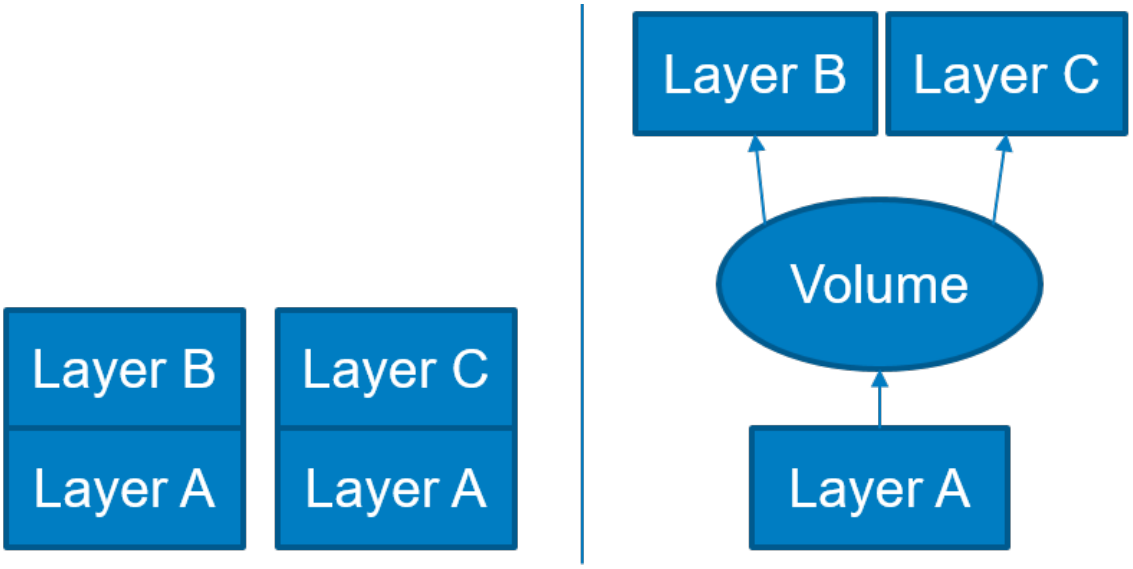

Figure 13. *Base layer vs configuration volume*

5. TEST SETUP

This chapter discusses the tests run with Docker containers to study if they are a feasible technology for embedded and/or RT systems.

5.1 Test System

Tests were run on WRM247+, Internet of Things (IoT) device with RT Linux. The system has a Cortex-A5 single core processor, 256 MiB of DDR2 memory and 256 MiB of NAND flash. Additionally, the machine has peripherals such as digital and analog IOs, USB buses, CAN, RS485, RS-232, Ethernet and SD card slot [9]. Before testing the system was running RT patched version of Linux 3.6.9 but this was replaced with RT patched Linux 4.9 since Docker suggests minimum Linux version of 3.10 [66]. The system also contains two OSs which made it possible to have one running OS download the new OS using network and flash it into the location of the other OS. This makes Over-the-air (OTA) updates possible. The flash reserved for each read-only root file system is 29 MiB. Additionally, the system has a read-write shared partition with a size of 182 MiB shared among the OSs. The shared partition is not flashed during update.

WRM247+ was chosen as the test platform as it was offered by Wapice and is used in real applications. Already existing RT Linux also supported this choice as Docker had not yet been studied with RT systems. WRM247+ is normally used for industrial Internet and IoT applications. Due to the lack of flash the test applications and Docker were placed on external SD card which was formatted to ext4. Docker additionally had problems with mounting overlayfs into SD card so mounts were placed in internal flash shared partition.

Docker was chosen as the container technology mainly due to its vast documentation and broad adoption in the industry. Docker's already existing Yocto recipe made the cross compilation easy. Many container technologies are also compatible with each other making the performance of different engines, in theory, quite similar.

5.2 Test Design

Embedded systems generally have limited resources, mainly CPU time, memory and non-volatile memory. Additionally, RT requirements enforce rule that the system has to react to stimuli in a predetermined time. Tests were designed to measure these critical properties to find out what kind of overhead containers introduce into the system.

To test the CPU latency produced by containers, POSIX IPC was chosen. POSIX IPC interface is simpler than that of System V's and with IPC it is possible to distribute variable

number of programs inside and outside of containers and compare the latencies and therefore find out the extra latency produced by containers. Up to 21 processes were executed simultaneously. This was chosen as top limit as embedded system generally have far less payload processes.

The interesting part of the memory consumption is the overhead produced by containers. Therefore, memory consumption was decided to be measured using the same tests as IPC latency but simultaneously logging the memory usage. Docker daemon memory consumption was also measured to find the overhead compared to execution in a system without Docker running.

File system usage was measured simply by checking the sizes of the files included in the system with Docker. To produce a realistic result hard links were discarded as two hard links pointing to a same file will both have the size of that file even though only one file exists in the file system. The test image size was compared with the sizes of files included in that image to check the extra driver usage of images.

5.3 Test Implementation

The first step of testing was to change the Yocto version used with the system from tag *Dylan* to tag *Morty*. This step was performed as Dylan does not contain a Docker recipe by default and the Linux versions were not officially supported by Docker. Other option for the Yocto tag was *Rocko* but this was not chosen due to issues with binutils 2.29 and Thumb2 instruction set [67]. The Linux version 4.9 was chosen for the measurements as it was the newest Linux version included in Morty by default and had an RT patch available. After Linux was configured correctly for Docker and Docker successfully executed on the device the Docker and its dependencies were moved to another image in Yocto which could be easily moved to the SD card attached to the device. The original locations of the relocated files were replaced with symbolic links[1] pointing to the new locations.

The core test application used for latency and memory testing consisted of one master process and multiple slave processes implemented in C that communicated with each other via named semaphores. The binaries were cross compiled, directly copied to the SD card and slave binary additionally placed in a Docker image with its dependencies and downloaded to the device via a Docker registry running on the development machine. The test application also had shell scripts used for running multiple tests sequentially. Memory test was identical to latency test except for one extra process running concurrently with lower priority logging the output of *free* to a log file.

Tests were run with slave counts of 1, 2, 3, 5, 10, 15 and 20 each with slaves executed either natively without containers, all slaves within a single container and with all slaves

[1] Symbolic link contains a path of another object.

placed in individual containers. This resulted in 20 different measurements as there were only two measurements with one slave. Master was always executed natively and posted all slave semaphores 100,000 times on each execution logging the time difference between posting the semaphores and consuming all semaphores posted by the slaves.

One iteration of the latency measurement can be seen in figure 14. As all processes are executing with SCHED_FIFO and RT priority of 99, the processes do not interrupt one another and processes need to voluntarily release the CPU i.e. the executing process is switched when the current process starts to wait for a semaphore.

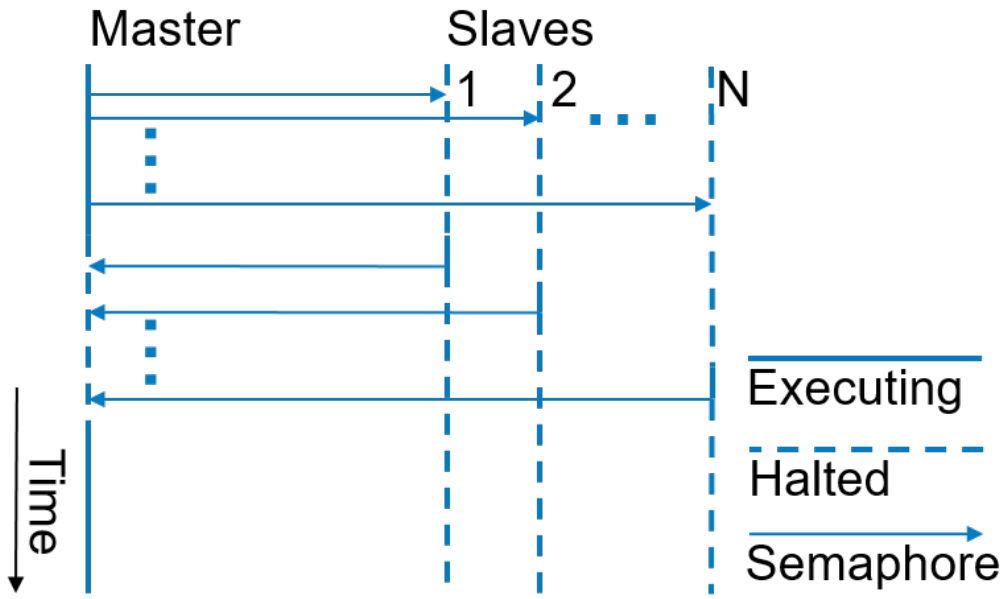

Figure 14. *Test execution*

5.3.1 Core Test Application

The main function of the master program can be found in program 10. Program takes all slave IDs as command line arguments and posts the respective semaphores. If no matching slave exists, the test will halt as master tries to wait its semaphore more times than it is posted stopping the process on line 23.

The main function of the slave program can be found from program 11. Similarly, to the master program, slave takes all slave IDs as arguments. Then parent slave process forks itself until all slave IDs can be distributed among programs. Slaves then proceed to wait for semaphores pointed by their IDs and simply post master's semaphore once they are executed. Starting multiple slaves would have been simpler using shell scripts but plain C implementation was chosen to keep the containers as basic as possible.

```
1  int main(int argc, char** argv){
2
3      // Init the test system.
4      int ret = init(argc, argv);
5      if(ret < 0){
6          exit(1);
7      }
8      int exec_count = MEAS_COUNT;
9
10     printf("Start Time\t\tStop Time\t\tDelta\n");
11
12     while(exec_count--){
13         // Store start time
14         setstarttime(&glatency);
15
16         int i;
17         // Post all slave semaphores
18         for(i=0;i<argc-1;i++){
19             shdsem_post(gmtos_sems[i]);
20         }
21         // Consume own semaphore until all slaves have posted
22         for(i=0;i<argc-1;i++){
23             shdsem_wait(gstom_sem);
24         }
25         // Store stop time
26         setstoptime(&glatency);
27         // Calculate delta
28         calcdiff(&glatency);
29
30         // Print result to stdout
31         printf("%lu\t%ld\t%lu\t%ld\t%lu\t%ld\n",
32           glatency.start.tv_sec,
33           glatency.start.tv_nsec,
34           glatency.stop.tv_sec,
35           glatency.stop.tv_nsec,
36           glatency.diff.tv_sec,
37           glatency.diff.tv_nsec);
38         nanosleep(&LOOPDELAY_NS, NULL);
39
40     }
41
42     do_cleanup(argc, argv);
43
44 }
```

Program 10. *Main function of the master*

```
1  int main(int argc, char** argv){
2
3      printf("Forking...\n");
4      do_forking(argc-1, &argv[1]);
5
6      printf("Initializing...\n");
7      int ret = init(gslave_id);
8
9      if(ret != 0){
10         printf("Init failed\n");
11         exit(1);
12     }
13     printf("Executing:\n");
14     while(1){
15         shdsem_wait(gmtos_sem);
16         // Exit if ctrl semaphore has been posted
17         if(!shdsem_trywait(gctrl_sem)){
18             break;
19         }
20         shdsem_post(gstom_sem);
21     }
22
23 }
```

Program 11. *Main function of the slave*

5.3.2 Test Scripts

Two BusyBox shell scripts were used to make testing the containers more automated [68]. The script used in latency testing can be found in appendix A and the memory test script in program 12. Formerly mentioned script simply executes tests with given number of slaves, e.g. *./exec_latency_test 1 2 3* would execute all tests with 1, 2 and 3 slaves and write log to different log files.

Dockerfile used to create the test image can be found in program 13. The image produced by the file contains the slave binary and all its dependencies listed by executing *objdump*. A simple Docker registry container was then setup on development Ubuntu and the image pushed there so it could be pulled from the embedded board.

```
1  #!/bin/sh
2
3  OUTPUT_FILE="mem_output.txt"
4
5
6  doLog() {
7      rm -f ${OUTPUT_FILE}
8      while true
9      do
10         free >> ${OUTPUT_FILE}
11         sleep 1
12     done
13 }
14
15
16
17 doLog &
18 LOG_PID="$!"
19 ./exec_latency_test.sh "$@"
20 kill ${LOG_PID}
```

Program 12. *Memory test shell script*

```
1  # Docker file for creating images
2  # for measuring overhead introduced by containers
3  #
4  # @author Toni Lammi toni.lammi@wapice.com
5
6  # Create a base image (not based on any other image)
7  FROM scratch
8
9  ENV PATH "/lib:/usr/lib"
10 # CPU latency slave, installed to root of the image
11 ADD slave /
12
13 # Linker and dynamic libraries
14 ADD ld-2.23.so /lib/ld-linux-armhf.so.3
15 ADD libc-2.23.so /usr/lib/libc.so.6
16 ADD libpthread-2.23.so /lib/libpthread.so.0
17
18 # Default entry point for the container
19 CMD ["/slave","0","1","2"]
```

Program 13. *Dockerfile used to build test containers*

6. RESULTS AND DISCUSSION

This chapter discusses the results of measurements described in chapter 5.

6.1 Inter-process Communication Latency

Measurement results were divided into four groups: min, mean, max and worst first latency. The first three do not include first latency measurement since it was consistently the largest latency. The worst first latency contains the longest first latency measurement of all test executions. Measurement results can be found in figures 15 and 16. The first shows the min, mean, max and worst first latencies of two slave test with 3 measurement executions, totaling in 300,000 iterations. The second is the same measurement but performed with 20 slaves.

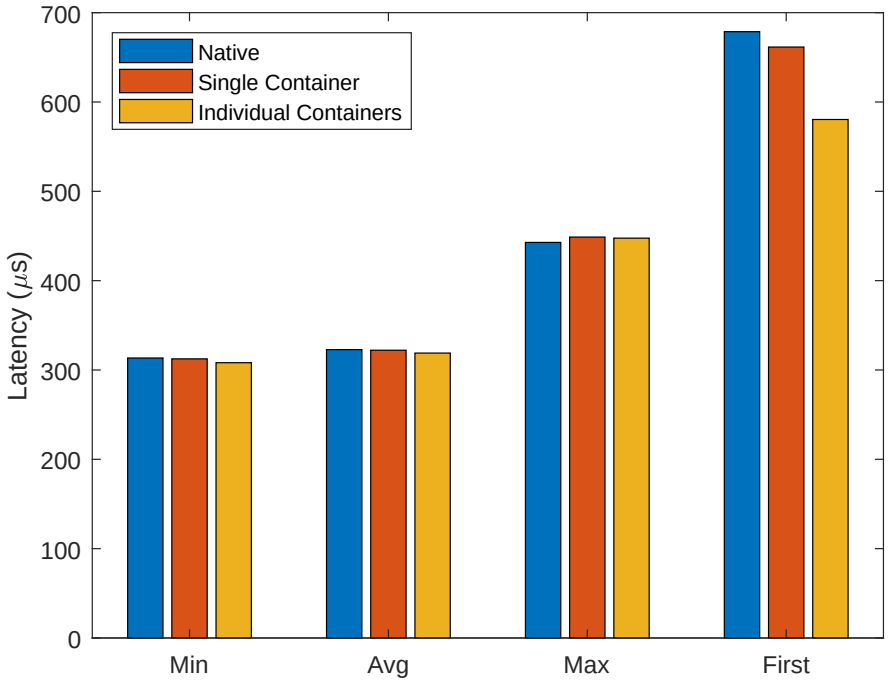

Figure 15. *Minimum, mean, max and start latency with two slaves*

The *first* groups in figures 15 and 16 are the worst first latencies when executing tests. As seen in the figures, with smaller slave counts the processes inside containers behave without notable difference to native processes. With larger slave counts, i.e. with 10, 15 and 20 slaves, executing slave processes in containers introduced long first measurement latencies as seen in figure 16.

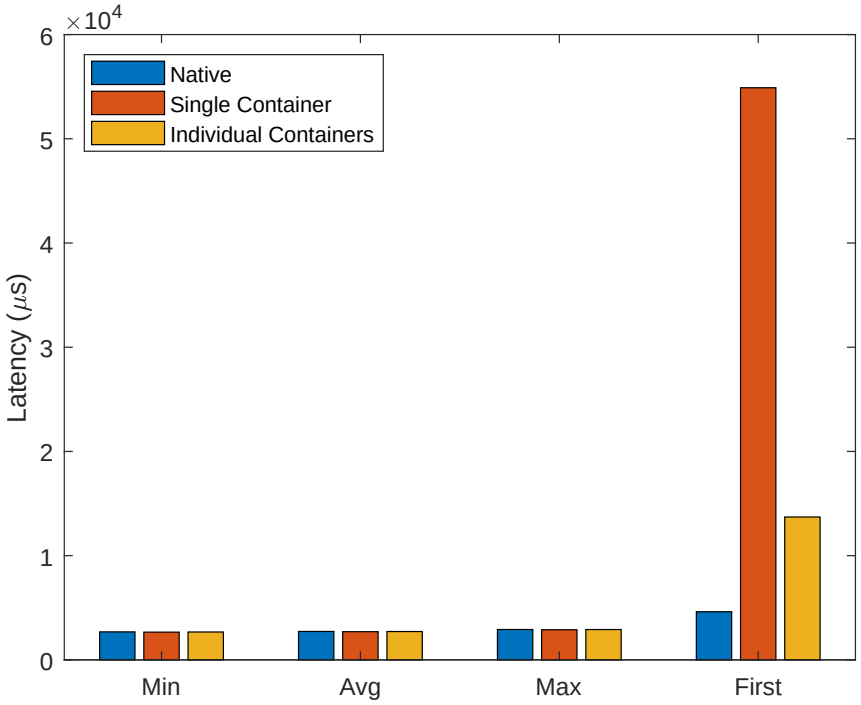

Figure 16. *Minimum, mean, max and start latency with 20 slaves*

The generally longer first latency might be due to Linux's caches but since there is no difference between native and container processes it is out of the scope of this thesis. For the extra-long first latencies I considered the container boot times. It might be that processes such as containerd-shim which inherit the priority of dockerd might not have enough time to bring up containers before the master is first started and therefore interfering with the measurements. To test this hypothesis I executed new tests after setting the dockerd's scheduling policy to SCHED_FIFO and setting RT priority to 99 which was done using command *chrt -f 99 $(pidof dockerd)*. New tests were executed using only ten iterations of latency measurements since the interesting part was only the first iteration of the latency measurement. Results of these measurements can be found in figures 17 and 18. Measurements were performed three times similarly to measurements discussed earlier in this chapter.

As seen in the figures, the considerably longer first latency disappeared. As the priority of dockerd was increased, also its child processes such as containerd-shims and runcs inherited this priority being therefore able to finish initialization of containers before the tests were initiated.

Figures 19, 20 and 21 contain test results for min, mean and max delay development as a function of slave count. Additionally, the parameters of functions gained by linear regression can be found in table 1. As seen in the table and figures, development of latency is highly linear with no large enough differences between test cases to state any differences in performance.

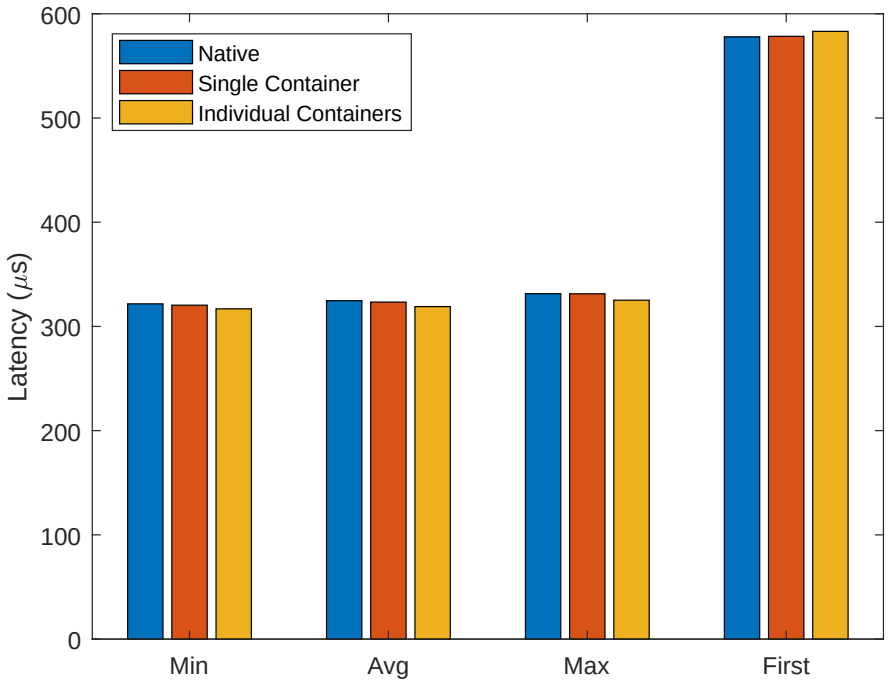

Figure 17. *IPC latencies with dockerd with RT priority and 2 slaves*

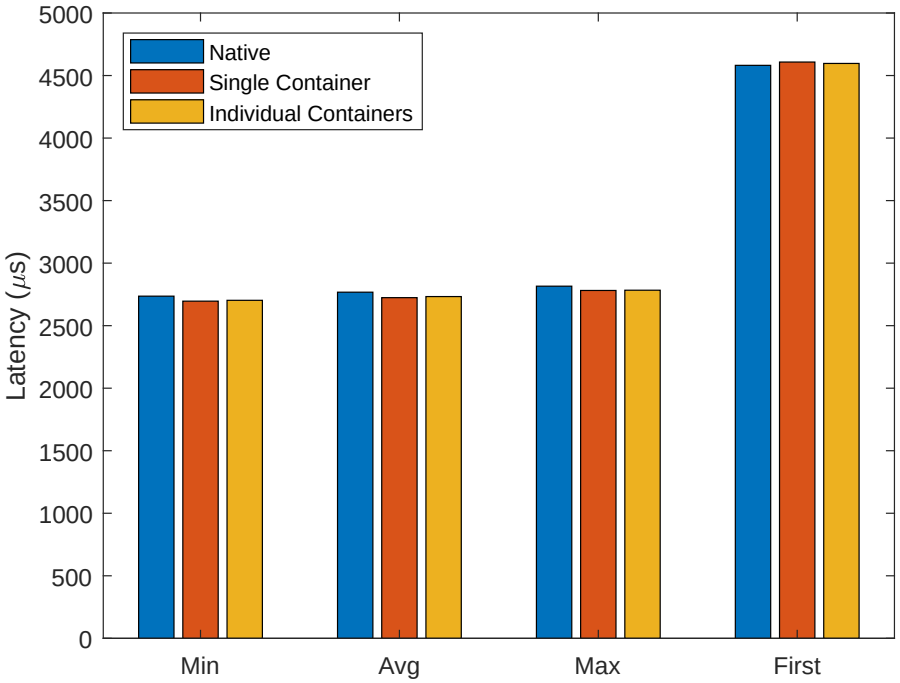

Figure 18. *IPC latencies with dockerd with RT priority and 20 slaves*

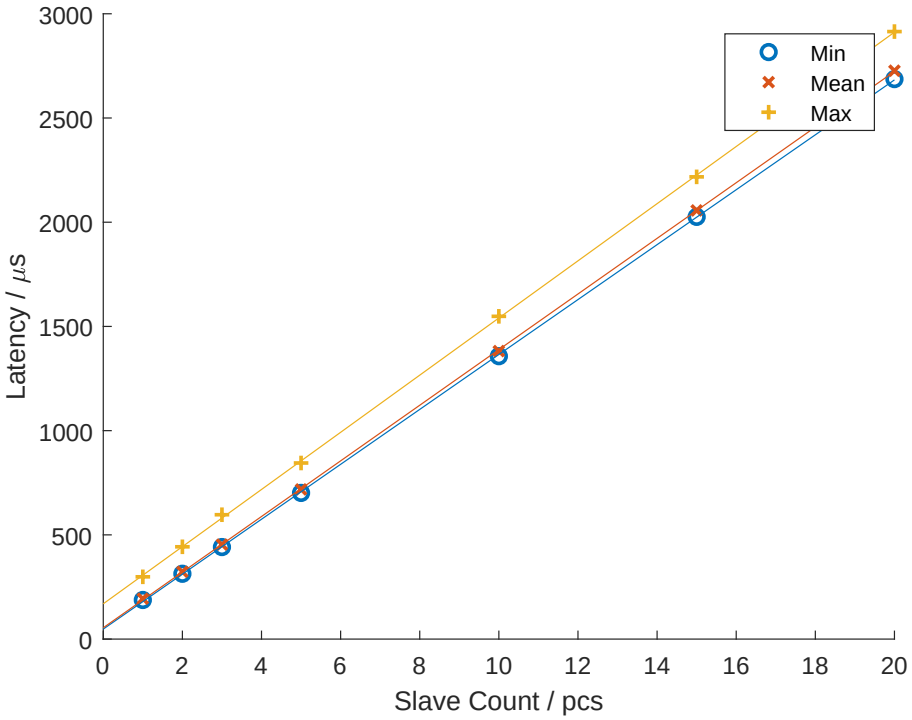

Figure 19. *Native execution latencies*

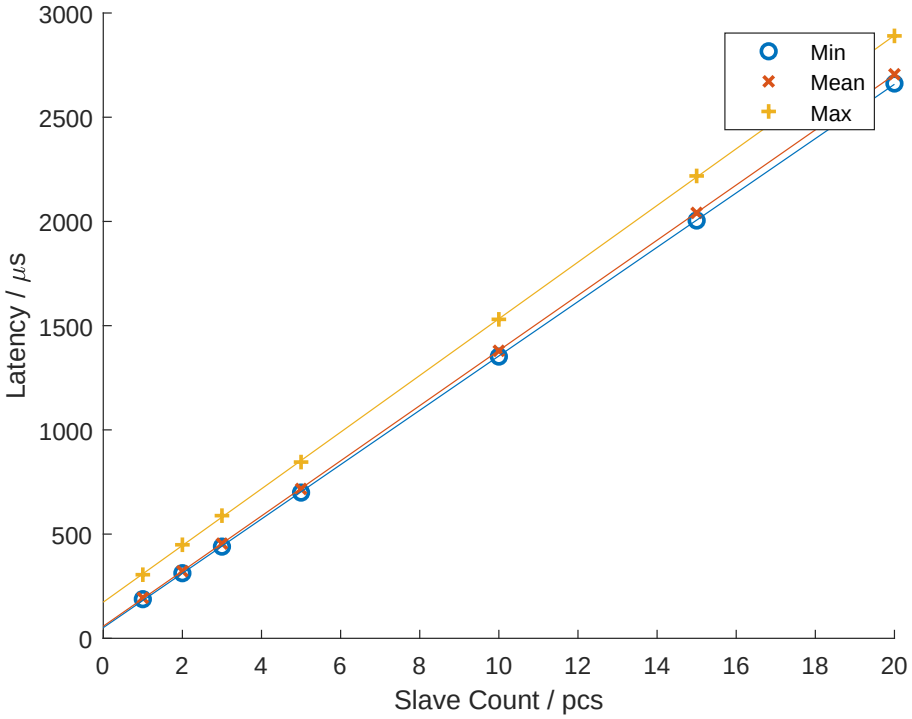

Figure 20. *Single container execution latencies*

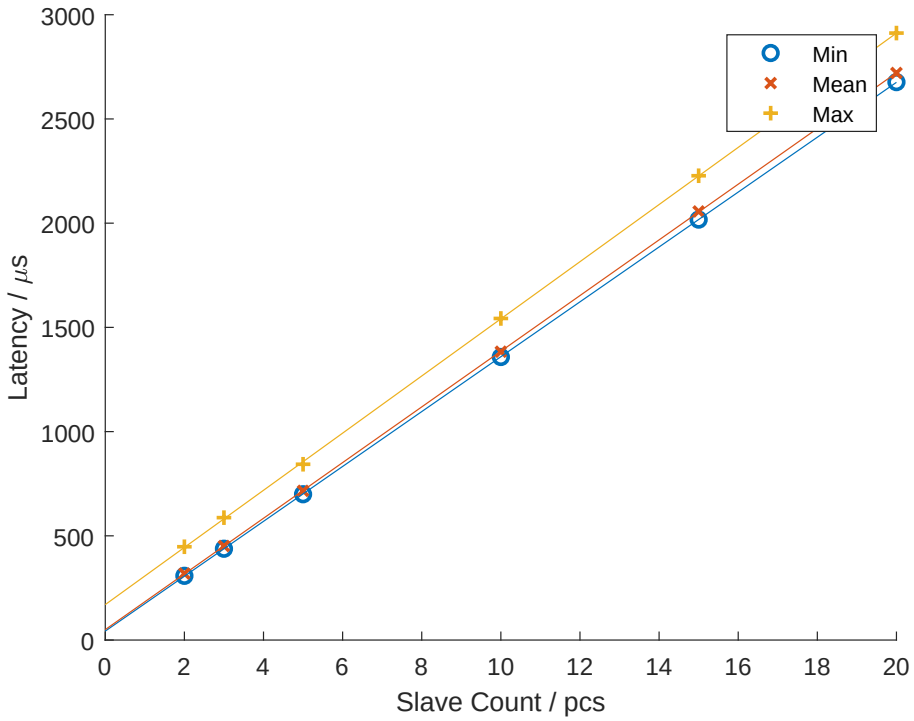

Figure 21. *Individual container latencies*

Table 1. *Values of offset and differential of linear regression in figures 19, 20 and 21*

y=ax+b	Test case	min	mean	max
a (μs/pcs)	native	131.7352	133.4428	137.1179
	single	130.3455	132.296	135.955
	individual	131.6335	133.5599	137.0892
b (μs)	native	48.0266	54.3472	169.2941
	single	51.1362	57.8912	173.3508
	individual	42.9437	49.9827	170.0959

6.2 Memory Usage

The memory consumption measurements were executed three times, each measurement execution consisting of 100,000 measurements totaling in 300,000 iterations. The development of free memory as the function of slave count can be found in figure 22. The figure contains three groups, each with different filling and three lines. The lines represent min, mean and max memory consumption of the measurements.

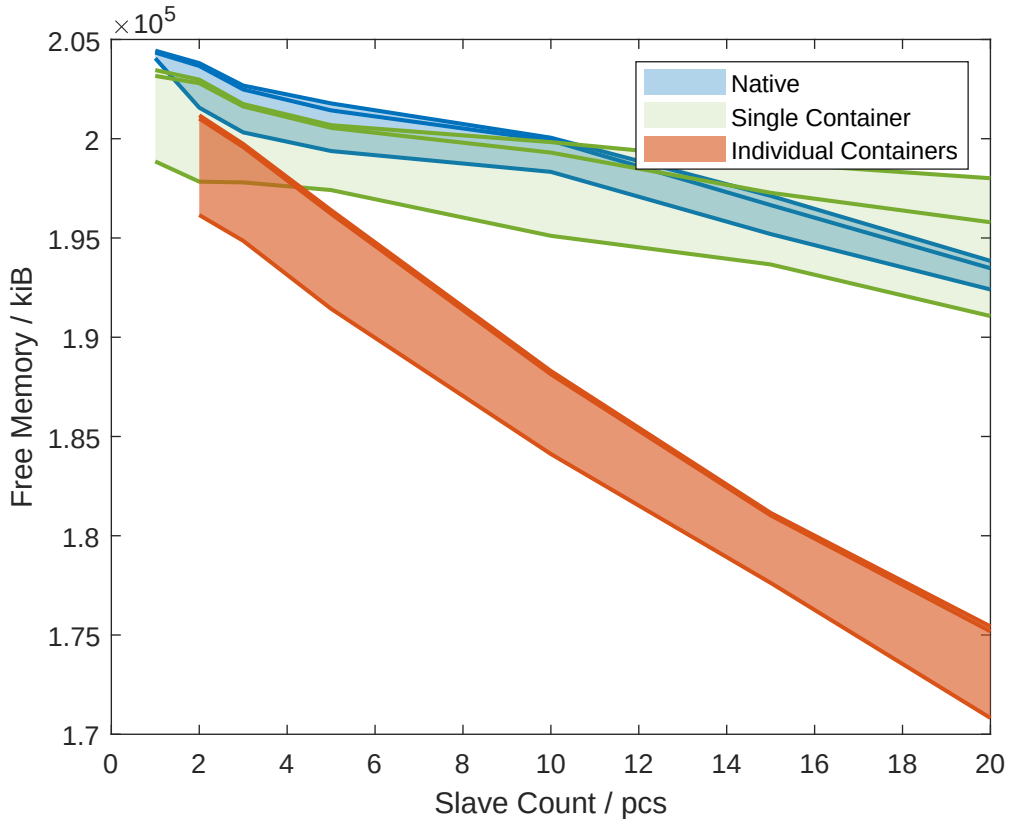

Figure 22. *Free memory with different measurement occasions.*

Distributing slave processes to individual containers consumes more memory than using single container or executing the test natively. Linear regression characteristic values of the mean memory usage can be found in table 2.

Table 2. *Characteristic values of linear regressions of memory measurements*

y=ax+b	a (KiB/slave)	b (MiB)
Native	-541	200
Single Container	-379	198
Individual Containers	-1462	199

After performing container memory consumption, the dockerd's memory usage was measured by toggling dockerd running and killed and logging the free memory. The results can be found in table 3.

Table 3. *Free memory with and without dockerd running*

	Docker Daemon Running	Docker Daemon Killed
Free Memory (MiB)	227	235

6.3 Mass Storage Usage

Mass storage usage of different binaries used by Docker can be found in table 4. *Stripped Total* field ignores values of containerd-ctr and docker-proxy as they can be excluded from the system without affecting the core functionality. The size of UBIFS[1] root file system produced by Yocto increased from *24.13 MiB* to *81.38 MiB* so the mass storage consumption over tripled.

Table 4. *Mass storage usage of different Docker binaries*

Binary Name	Size (MiB)
docker	11.12
docker-proxy	2.27
dockerd	33.16
containerd	12.95
containerd-ctr	12.41
containerd-shim	6.83
runc	10.79
Total	89.53
Stripped Total	74.85

Based on the table above we can state that deploying Docker to an embedded Linux consumes considerable amount of storage. The files included in test Docker image had total size of 1112892 bytes gained with *ls -l*. The size of the image produced was 11.1 MB according to *docker images* command so image does not consume much extra storage. Image and container size is however quite complex subject and can be greatly affected by how these features are used. For example, using one base image for multiple child images can save space.

6.4 Container Measurements in Other Studies

Several studies considering Docker on servers and other non-embedded systems have been concluded and since the functional principles of these machines are similar they can be used to give some idea of the overhead introduced by Docker and containers in general in embedded systems. This section introduces and discusses six of these studies and their results.

Di Tomaso et. al compared using Docker container-based solution instead of native programs when calculating performance of genomic pipelines [69]. The study found out that

[1] File system optimized for unmanaged flash storage.

with shorter tasks the containers were clearly worse solution than native execution but when the execution time grew the performance of containers got closer and closer to native binaries. The measurements included start up times of the containers so containers performed almost identically to native software if the start up time is ignored.

Felter et. al studied the overhead of VMs and Docker with different settings when running an SQL server inside the container [70]. The measurements show that Docker is a lot faster solution in running SQL server than KVM. The same measurements also show a considerable drop in throughput when aufs is used as storage instead of volumes. Using network namespaces produces minimal overhead compared to containers that share the host network namespace.

A study by Arango et. al. studied different container technologies and compared their performance to natively run software [71]. The study found out that bind mounts perform better than unioning file system in both, reading and writing, but difference in writing was especially large: the bind mount write was approximately 2.5 times as fast. When random write and read were tested the difference was smaller but still the bind mount operations were clearly faster. The largest overhead produced by Docker was measured with network bandwidth where containers presented 16.96% higher bandwidth usage than native software. Otherwise the overhead introduced by Docker when reading and writing bind mounts was very minimal and performance was close to native processes. Similar study performed in 2013 studied performance of LXC against other technologies [72]. This study got similar results to former study but also inspected the performance of the test system under fork bomb attack which managed to bring LXC solution down but no other technologies.

M.T. Chung et. al. studied the overhead of Docker and compared it to overhead of VMs [73]. The results were again expected, and Docker generally produces minimal overhead compared to VMs when executing distributed computing with high intensity. With higher machine counts VM and container performances get closer to each other as an increasing amount of resources is used for the task distribution instead of the actual computing.

Lastly, Avino et. al. measured Docker performance when containers run multimedia and gaming servers [74]. This study was the only one to achieve results were the overhead of Docker was not minimal. Multimedia servers introduced overhead of 45% when one client was served but when the client count was raised to eight the overhead dropped to under 10%. The respective overheads were 3% and less than 2% on gaming servers. The exceptionally high overheads could be explained e.g. by badly optimized containers where file operations are performed in unioning file system since container execution is not well documented.

The studies discussed in this section generally provided similar results as this study. Only study which noticed a considerable overhead with containers did not explain the configuration of containers. Non-optimized configuration could explain at least some of the overhead.

6.5 Information Security

Docker containers are a good addition from information security point of view. They provide an easy way of hardening the system and can be further enhanced with MAC programs such as AppArmor. This security however should not be blindly trusted as attacker gaining access to a container can still break functionality of the container and other depending containers. Containers are also not as well isolated from the host as e.g. VMs. Some known Docker vulnerabilities are listed by Common Vulnerabilities and Exposures (CVE) and the list seems to have more listings for older Docker versions than newer [75]. Therefore, the Docker version used should always be the newest possible. This is somewhat a problem as updating Docker for a custom device is not as straight forward as for a Ubuntu running on AMD64 architecture.

6.6 System Deployment and Software Management

Tools provided by Docker help in deploying the systems. Overlay networks can pack groups of dockerds into clusters and make it easier to address whole groups e.g. using only one IP address. The autonomous containers then again make it possible to update the Docker version while the containers are not affected in any way and as the applications are executed in their own sandboxes the developers can design systems without having to consider dependencies among applications as much as without Docker.

Docker provides a good way of managing software packages where each application can be packaged into a sandbox or a group of sandboxes. This is a lot faster compared to flashing a completely new firmware into the device and does not have same issues as with Debian packages with read-only file systems. Containers also solve software dependency issues which may exist with Debian packages and if an invalid Docker image is downloaded the problem can be easily fixed by downloading a working version.

Docker container deployment can be automated using tools such as open-horizon which only requires that images are pushed to a deployment server from which the deployment to machines works automatically. New images also do not have to be completely downloaded into the devices, but it is enough to download only the layers which do not already exist in the device therefore saving bandwidth.

The build system would also require an update when software is deployed in containers instead of natively. At the beginning the development process works similarly to normal development but after the software reaches a point where it can be taken into use the software should be containerized, first manually. After the recipe for the image is functional

additional steps can be added to normal build to pack the software into images and push them to registries. In the case of production images, a notification should also be sent to get the software into the devices. This obviously does not have to be done by build software and can be done e.g. by the server handling the deployment.

6.7 Docker and the Test System

The test system is not a feasible platform for Docker as it is due to larger flash usage of the root file system. This could however be fixed quite easily by increasing the flash reserved for the file systems at the expense of shared partition. This would result in approximately 85 MiB partitions for root file systems and a 72 MiB shared partition. Docker then needs a read-write partition for storing images and containers so the shared partition should either be further divided into two or part of the partition should be used by Docker.

The major negative effect of this change would be that the layout of the flash would change. This means that OTA updates are not simple between Docker and non-Docker systems as the bootloader has to be reconfigured for the new OS. The smaller con is that the storage left for e.g. measurements gets smaller (shared partition is used for this since it is a read-write partition) and therefore, data might have to be stored to SD card. In case of critical data an encryption mechanism should be provided for the file system.

Finally, the cost of deploying Docker should be considered. The obvious benefits are better information security and software management but the implementation of Docker would require changes to the update system and deploying new system for storing the Docker images for downloading into devices.

7. CONCLUSIONS AND FUTURE WORK

The objective of this thesis was to determine if containers and especially Docker are a feasible technology for embedded and RT Linux systems and what would be the advantages and disadvantages of adopting the technology. The thesis first discussed the underlying Linux features used by containers followed by more detailed description of Docker features and information security and how to deploy Docker to an embedded system. The next chapters discussed the design and results of measurements of Docker containers as well as analyzed the changes required for the test system to deploy Docker. Measurements and measurement results from other studies about Docker and other container engines were also discussed.

From the measurements performed in this thesis and in other studies we can conclude that performance-wise Docker containers are a feasible solution to be used in embedded and RT Linux systems if the system has enough mass storage capacity and memory and Docker images used with the programs are optimized. For comfortable use of Docker some guidelines could be 10 MiB extra memory compared to native need and extra 80 MiB of mass storage space. No notable difference was found with IPC latencies in containerized and native applications. For the system used in tests Docker is a possible technology but requires changes to be made to the system. Docker could be executed in the device, but this would need a flash layout change which would make the new firmware incompatible with older bootloader configurations and vice versa. The complete analysis of the feasibility would require analysis of required work compared to benefits gained with Docker.

It was also found that Docker helps in deploying new software or updating old software which can be done e.g. with help of additional processes. Docker was found to help with information security as processes running in containers are isolated from the rest of the system. Containers should however not to be blindly trusted as the protection is not perfect. To help with the security user could e.g. harden the system by applying AppArmor to the system and handle the device permissions to the cloud so that compromised devices do not have any extra permissions and all permissions are as easy to revoke as possible. The usergroup *docker* should not be used to prevent privilege escalation.

The natural way of continuing the study would be to use different container engines, like Balena and rkt, for similar tests and check the difference in their performance. The other feature which should be scrutinized is execution of more optimized program both natively and in container and inspecting these results. In this context a more optimized would mean that e.g. no inter-process communication is used as it is a lot slower than using only inter-thread communication. If containers were to be taken into use, some guidelines

should also be created for development and deployment. This would require development of a build system which can automatically create and push images to registries. Also, some training would be required for embedded developers not previously familiar with containers.

REFERENCES

[1] Run multiple services in a container, docs.docker.com. Available (accessed on 8.5.2018): https://docs.docker.com/config/containers/multi-service_container/

[2] About, www.opencontainers.org. Available (accessed on 8.5.2018): https://www.opencontainers.org/about

[3] Docker Github, Github. Available (accessed on 7.6.2018): https://github.com/docker/docker-ce

[4] What's LXC?, linuxcontainers.org. Available (accessed on 30.3.2018): https://linuxcontainers.org/lxd/introduction/

[5] LXC – Linux Containers, github.com. Available (accessed on 30.3.2018): https://github.com/lxc

[6] J. Nickoloff, Docker in Action, 1st ed., Manning Publications Co., Greenwich, CT, USA, 2016, 274 p.

[7] balena, www.balena.io. Available (accessed on 18.5.2018): https://www.balena.io/

[8] rkt, coreos.com. Available (accessed on 18.5.2018): https://coreos.com/rkt/

[9] WRM 247+, www.iot-ticket.com. Available (accessed on 30.3.2018): https://www.iot-ticket.com/images/Files/WRM247+_DataSheet_2015.pdf

[10] Android Kernel, www.android.com. Available (accessed on 19.5.2018): https://source.android.com/security/

[11] Usage of operating systems for websites, w3techs.com. Available (accessed on 19.5.2018): https://w3techs.com/technologies/overview/operating_system/all

[12] Operating System Family / Linux, top500.org. Available (accessed on 8.5.2018): https://www.top500.org/statistics/details/osfam/1

[13] cgroups – Linux control groups, man7.org. Available (accessed on 16.3.2018): http://man7.org/linux/man-pages/man7/cgroups.7.html

[14] Control Group v2, Github. Available (accessed on 12.5.2018): https://github.com/torvalds/linux/blob/v4.16/Documentation/cgroup-v2.txt

[15] Linux source code namespace management, Github. Available (accessed on 14.3.2018): https://github.com/torvalds/linux/blob/v4.16-rc5/include/linux/nsproxy.h

[16] W. Mauerer, Professional Linux Kernel Architecture, Wiley Publishing, Inc., 2008, 1116 p. Available: https://cse.yeditepe.edu.tr/~kserdaroglu/spring2014/cse331/termproject/BOOKS/ProfessionalLinuxKernelArchitecture-WolfgangMauerer.pdf

[17] Linux source code namespace creation, Github. Available (accessed on 2.6.2018): https://github.com/torvalds/linux/blob/v4.16/kernel/nsproxy.c

[18] clone, __clone2 – create a child process, man7.org. Available (accessed on 9.5.2018): http://man7.org/linux/man-pages/man2/clone.2.html

[19] unshare – disassociate parts of the process execution context, man7.org. Available (accessed on 9.5.2018): http://man7.org/linux/man-pages/man2/unshare.2.html

[20] copy_process(), Github. Available (accessed on 9.5.2018): https://github.com/torvalds/linux/blob/v4.16/kernel/fork.c

[21] Namespaces in operation, part 1: namespaces overview, lwn.net. Available (accessed on 18.3.2018): https://lwn.net/Articles/531114/

[22] namespaces – overview of Linux namespaces, man7.org. Available (accessed on 27.8.2018): http://man7.org/linux/man-pages/man7/namespaces.7.html

[23] user_namespaces – overview of Linux user namespaces, man7.org. Available (accessed on 10.5.2018): http://man7.org/linux/man-pages/man7/user_namespaces.7.html

[24] capabilities – overview of Linux capabilities, man7.org. Available (accessed on 2.8.2018): http://man7.org/linux/man-pages/man7/capabilities.7.html

[25] Mount namespaces and shared subtrees, lwn.net. Available (accessed on 23.3.2018): https://lwn.net/Articles/689856/

[26] cgroup_namespaces – overview of Linux cgroup namespaces, man7.org. Available (accessed on 16.3.2018): http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html

[27] network_namespaces – overview of Linux cgroup namespaces, man7.org. Available (accessed on 19.7.2018): http://man7.org/linux/man-pages/man7/network_namespaces.7.html

[28] Linux Security Module Usage, www.kernel.org. Available (accessed on 19.5.2018): https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html

[29] AppArmor Wiki, www.gitlab.com. Available (accessed on 19.5.2018): https://gitlab.com/apparmor/apparmor/wikis/GettingStarted

[30] Core kernel scheduler code and related syscalls, Github. Available (accessed on 17.4.2018): https://github.com/torvalds/linux/blob/master/kernel/sched/core.c

[31] Scheduler internal types and methods, Github. Available (accessed on 17.4.2018): https://github.com/torvalds/linux/blob/master/kernel/sched/sched.h

[32] prio.h, Github. Available (accessed on 6.6.2018): https://github.com/torvalds/linux/blob/v4.16/include/linux/sched/prio.h

[33] Preempt-RT history, wiki.linuxfoundation.org. Available (accessed on 2.6.2018): https://wiki.linuxfoundation.org/realtime/rtl/blog#preempt-rt-history

[34] Real-Time Linux source code, Github. Available (accessed on 11.5.2018): https://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git/about/

[35] The Go Programming Language, golang.org. Available (accessed on 6.7.2018): https://golang.org

[36] Docker overview, docs.docker.com. Available (accessed on 7.5.2018): https://docs.docker.com/engine/docker-overview/

[37] Overview of Docker editions, docs.docker.com. Available (accessed on 7.5.2018): https://docs.docker.com/install/overview/

[38] containerd, containerd.io. Available (accessed on 7.5.2018): https://containerd.io/

[39] Containerd 0.2 UnionFS, github.com. Available (accessed on 20.5.2018): https://github.com/containerd/containerd/blob/v0.2.0/docs/bundle.md#root-filesystem

[40] Containerd 1.1 UnionFS, github.com. Available (accessed on 20.5.2018): https://github.com/containerd/containerd/tree/release/1.1#root-filesystems

[41] Spinning Out Docker's Plumbing: Part 1: Introducing runC, blog.docker.com. Available (accessed on 10.6.2018): https://blog.docker.com/2015/06/runc/

[42] Overlay Filesystem, www.kernel.org. Available (accessed on 6.8.2018): https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

[43] Use the OverlayFS storage driver, docs.docker.com. Available (accessed on 3.9.2018): https://docs.docker.com/storage/storagedriver/overlayfs-driver/

[44] Manage data in Docker, docs.docker.com. Available (accessed on 3.5.2018): https://docs.docker.com/storage/

[45] Use volumes, docs.docker.com. Available (accessed on 3.5.2018): https://docs.docker.com/storage/volumes/

[46] Use bind mounts, docs.docker.com. Available (accessed on 3.5.2018): https://docs.docker.com/storage/bind-mounts/

[47] Use tmpf mounts, docs.docker.com. Available (accessed on 3.5.2018): https://docs.docker.com/storage/tmpfs/

[48] Limit a container's resources, docs.docker.com. Available (accessed on 3.5.2018): https://docs.docker.com/config/containers/resource_constraints/

[49] support cgroup v2, github.com. Available (accessed on 1.6.2018): https://github.com/opencontainers/runc/issues/654

[50] Networking Overview, docs.docker.com. Available (accessed on 8.5.2018): https://docs.docker.com/network/

[51] Use bridge networks, docs.docker.com. Available (accessed on 8.5.2018): https://docs.docker.com/network/bridge/

[52] Use Macvlan networks, docs.docker.com. Available (accessed on 8.5.2018): https://docs.docker.com/network/macvlan/

[53] Use host networking, docs.docker.com. Available (accessed on 8.5.2018): https://docs.docker.com/network/host/

[54] Use overlay networks, docs.docker.com. Available (accessed on 8.5.2018): https://docs.docker.com/network/overlay/

[55] Networking with standalone containers, docs.docker.com. Available (accessed on 8.5.2018): https://docs.docker.com/network/network-tutorial-standalone/

[56] Docker Image Specification v1.2, Github. Available (accessed on 2.6.2018): https://github.com/moby/moby/blob/17.05.x/image/spec/v1.2.md

[57] Getting started, Part 4: Swarms, docs.docker.com. Available (accessed on 30.3.2018): https://docs.docker.com/get-started/part4/

[58] Isolate containers wieh a user namespace, docs.docker.com. Available (accessed on 6.8.2018): https://docs.docker.com/engine/security/userns-remap/

[59] AppArmor security profile for Docker, docs.docker.com. Available (accessed on 6.8.2018): https://docs.docker.com/engine/security/apparmor/

[60] Deploy a registry server, docs.docker.com. Available (accessed on 2.8.2018): https://docs.docker.com/registry/deploying/

[61] docker login, docs.docker.com. Available (accessed on 2.8.2018): https://docs.docker.com/engine/reference/commandline/login/

[62] Yocto Project Overview and Concepts Manual, yoctoproject.org. Available (accessed on 8.6.2018): https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html

[63] Yocto Recipe of Docker, git.yoctoproject.org. Available (accessed on 20.5.2018): https://git.yoctoproject.org/cgit/cgit.cgi/meta-virtualization/tree/recipes-containers/docker

[64] dockerd, docs.docker.com. Available (accessed on 3.8.2018): https://docs.docker.com/engine/reference/commandline/dockerd/

[65] open-horizon github, Github. Available (accessed on 8.6.2018): https://github.com/open-horizon

[66] Docker frequently asked questions, docs.docker.com. Available (accessed on 4.5.2018): https://docs.docker.com/engine/faq/#what-platforms-does-docker-run-on

[67] [yocto] binutils 2.29.1 ARM Thumb kernel problem, yoctoproject.org. Available (accessed on 29.6.2018): https://lists.yoctoproject.org/pipermail/yocto/2018-April/040648.html

[68] BusyBox: The Swiss Army Knife of Embedded Linux, busybox.net. Available (accessed on 6.7.2018): https://busybox.net/about.html

[69] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M.L. Heuer, C. Notredame, The impact of Docker containers on the performance of genomic pipelines, PeerJ, Vol. 3, Sept. 2015, p. e1273.

[70] W. Felter, A. Ferreira, R. Rajamony, J. Rubio, An updated performance comparison of virtual machines and Linux containers, March, 2015, pp. 171–172.

[71] C. Arango, R. Dernat, J. Sanabria, Performance Evaluation of Container-based Virtualization for High Performance Computing Environments, CoRR, Vol. abs/1709.10140, 2017.

[72] M.G. Xavier, M.V. Neves, F.D. Rossi, T.C. Ferreto, T. Lange, C.A.F.D. Rose, Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments, Feb, 2013, pp. 233–240.

[73] M.T. Chung, N. Quang-Hung, M.T. Nguyen, N. Thoai, Using Docker in high performance computing applications, July, 2016, pp. 52–57.

[74] G. Avino, M. Malinverno, F. Malandrino, C. Casetti, C. Chiasserini, Characterizing Docker Overhead in Mobile Edge Computing Scenarios, CoRR, Vol. abs/1801.08843, 2018.

[75] Docker Security Vulnerabilities, cvedetails.com. Available (accessed on 29.6.2018): https://www.cvedetails.com/vulnerability-list/vendor_id-13534/product_id-28125/Docker-Docker.html

APPENDIX A: LATENCY TEST SHELL SCRIPT

```
1  #!/bin/sh
2
3  # Binary names
4  MASTER_BIN="./master"
5  SLAVE_BIN="./slave"
6
7  LOG_FILE_postfix=".txt"
8
9  LOG_FILE_native_prefix="native"
10 LOG_FILE_single_prefix="cont_single"
11 LOG_FILE_ind_prefix="cont_ind"
12
13
14 # run -d --ipc=host --cap-add=sys_nice \
15 #       --ulimit rtprio=99 10.165.163.188:5000/latencytest:initial
16 DOCKER_FLAGS="-d --ipc=host --cap-add=sys_nice --ulimit rtprio=99"
17 DOCKER_LATENCY_TEST_IMAGE="10.165.163.188:5000/latencytest"
18 DOCKER_ENTRY_POINT="/slave"
19
20 rmContainers() {
21     echo "Removing containers..."
22     docker rm $(docker ps -aq)
23     echo "Containers removed"
24 }
25
26 execNative() {
27     ${SLAVE_BIN}  $(seq 1 "$1")&
28     sleep 2
29     ${MASTER_BIN} $(seq 1 "$1")>\
30         ${LOG_FILE_native_prefix}${1}${LOG_FILE_postfix}
31     sleep 2
32 }
33
34 execSingle() {
35     docker run ${DOCKER_FLAGS} ${DOCKER_LATENCY_TEST_IMAGE}\
36         ${DOCKER_ENTRY_POINT} $(seq 1 "$1")
37
38     ${MASTER_BIN} $(seq 1 "$1") >\
39         ${LOG_FILE_single_prefix}${1}${LOG_FILE_postfix}
40     sleep 10
41
42     rmContainers
43 }
44
```

```
45 execIndividual() {
46     for i in $(seq 1 "$1")
47     do
48         docker run ${DOCKER_FLAGS} ${DOCKER_LATENCY_TEST_IMAGE}\
49                         ${DOCKER_ENTRY_POINT} $i
50
51     done
52
53     ${MASTER_BIN} $(seq 1 "$1") >\
54         ${LOG_FILE_ind_prefix}${1}${LOG_FILE_postfix}
55     sleep 10
56
57     rmContainers
58
59 }
60
61
62 echo "Executing tests"
63
64 for i in "$@"
65 do
66     echo "Testing with $i slaves"
67     echo "Running native tests"
68     execNative $i
69     sleep 2
70     echo "Running test in single container"
71     execSingle $i
72
73     if [ "$i" -gt 1 ]
74     then
75         echo "Running test in individual containers"
76         execIndividual $i
77
78     fi
79 done
80
81 echo "Finished"
```

APPENDIX B: CHECK-CONFIG.SH OUTPUT

```
1  Generally Necessary:
2  - cgroup hierarchy: properly mounted [/sys/fs/cgroup]
3  - apparmor: enabled and tools installed
4  - CONFIG_NAMESPACES: enabled
5  - CONFIG_NET_NS: enabled
6  - CONFIG_PID_NS: enabled
7  - CONFIG_IPC_NS: enabled
8  - CONFIG_UTS_NS: enabled
9  - CONFIG_CGROUPS: enabled
10 - CONFIG_CGROUP_CPUACCT: enabled
11 - CONFIG_CGROUP_DEVICE: enabled
12 - CONFIG_CGROUP_FREEZER: enabled
13 - CONFIG_CGROUP_SCHED: enabled
14 - CONFIG_CPUSETS: enabled
15 - CONFIG_MEMCG: enabled
16 - CONFIG_KEYS: enabled
17 - CONFIG_VETH: enabled
18 - CONFIG_BRIDGE: enabled
19 - CONFIG_BRIDGE_NETFILTER: enabled
20 - CONFIG_NF_NAT_IPV4: enabled
21 - CONFIG_IP_NF_FILTER: enabled
22 - CONFIG_IP_NF_TARGET_MASQUERADE: enabled
23 - CONFIG_NETFILTER_XT_MATCH_ADDRTYPE: enabled
24 - CONFIG_NETFILTER_XT_MATCH_CONNTRACK: enabled
25 - CONFIG_NETFILTER_XT_MATCH_IPVS: enabled
26 - CONFIG_IP_NF_NAT: enabled
27 - CONFIG_NF_NAT: enabled
28 - CONFIG_NF_NAT_NEEDED: enabled
29 - CONFIG_POSIX_MQUEUE: enabled
30
31 Optional Features:
32 - CONFIG_USER_NS: missing
33 - CONFIG_SECCOMP: missing
34 - CONFIG_CGROUP_PIDS: missing
35 - CONFIG_MEMCG_SWAP: missing
36 - CONFIG_MEMCG_SWAP_ENABLED: missing
37 - CONFIG_BLK_CGROUP: missing
38 - CONFIG_BLK_DEV_THROTTLING: missing
39 - CONFIG_IOSCHED_CFQ: missing
40 - CONFIG_CFQ_GROUP_IOSCHED: missing
41 - CONFIG_CGROUP_PERF: missing
42 - CONFIG_CGROUP_HUGETLB: missing
43 - CONFIG_NET_CLS_CGROUP: missing
44 - CONFIG_CGROUP_NET_PRIO: missing
```

```
45 - CONFIG_CFS_BANDWIDTH: missing
46 - CONFIG_FAIR_GROUP_SCHED: missing
47 - CONFIG_RT_GROUP_SCHED: enabled
48 - CONFIG_IP_VS: missing
49 - CONFIG_IP_VS_NFCT: missing
50 - CONFIG_IP_VS_RR: missing
51 - CONFIG_EXT4_FS: enabled
52 - CONFIG_EXT4_FS_POSIX_ACL: missing
53 - CONFIG_EXT4_FS_SECURITY: missing
54 enable these ext4 configs if you are using ext3 or ext4 as backing filesystem
55 - Network Drivers:
56 - "overlay":
57 - CONFIG_VXLAN: missing
58 Optional (for encrypted networks):
59 - CONFIG_CRYPTO: enabled
60 - CONFIG_CRYPTO_AEAD: enabled
61 - CONFIG_CRYPTO_GCM: enabled
62 - CONFIG_CRYPTO_SEQIV: enabled
63 - CONFIG_CRYPTO_GHASH: enabled
64 - CONFIG_XFRM: enabled
65 - CONFIG_XFRM_USER: enabled (as module)
66 - CONFIG_XFRM_ALGO: enabled (as module)
67 - CONFIG_INET_ESP: enabled (as module)
68 - CONFIG_INET_XFRM_MODE_TRANSPORT: enabled (as module)
69 - "ipvlan":
70 - CONFIG_IPVLAN: missing
71 - "macvlan":
72 - CONFIG_MACVLAN: missing
73 - CONFIG_DUMMY: missing
74 - "ftp,tftp client in container":
75 - CONFIG_NF_NAT_FTP: missing
76 - CONFIG_NF_CONNTRACK_FTP: missing
77 - CONFIG_NF_NAT_TFTP: missing
78 - CONFIG_NF_CONNTRACK_TFTP: missing
79 - Storage Drivers:
80 - "aufs":
81 - CONFIG_AUFS_FS: enabled
82 - "btrfs":
83 - CONFIG_BTRFS_FS: missing
84 - CONFIG_BTRFS_FS_POSIX_ACL: missing
85 - "devicemapper":
86 - CONFIG_BLK_DEV_DM: enabled
87 - CONFIG_DM_THIN_PROVISIONING: missing
88 - "overlay":
89 - CONFIG_OVERLAY_FS: enabled
90 - "zfs":
91 - /dev/zfs: missing
92 - zfs command: missing
93 - zpool command: missing
94
95 Limits:
```

```
96 - /proc/sys/kernel/keys/root_maxkeys: 1000000
```